

IEEE Std 1003.1™, 2004 Edition

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

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

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

System Interfaces

Sponsor

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

and

The Open Group



THE *Open* GROUP

[This page intentionally left blank]

Abstract

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

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

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

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

The following areas are outside the scope of this standard:

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

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

Keywords

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

Copyright © 2001-2004 by the Institute of Electrical and Electronics Engineers, Inc. and The Open Group. All rights reserved.

System Interfaces, Issue 6

Published 30 April 2004 by the Institute of Electrical and Electronics Engineers, Inc.

3 Park Avenue, New York, NY 10016-5997, U.S.A.

ISBN: 0-7381-4041-4/SH95235 PDF 0-7381-4042-2/SS95235

Printed in the United States of America by the IEEE.

Published 30 April 2004 by The Open Group

Apex Plaza, Forbury Road, Reading, Berkshire RG1 1AX, U.K.

Document Number: C047

ISBN: 1-931624-44-5

Printed in the U.K. by The Open Group.

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

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

Permissions

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

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

Feedback

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

IEEE

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

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

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

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

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

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

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

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

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>.

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

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

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

The Open Group

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

Further information on The Open Group is available at www.opengroup.org.

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

More information is available at www.opengroup.org/testing.

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

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

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

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published at www.opengroup.org/corrigenda.

The Open Group publications catalog and ordering information are available at 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	30	
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	36	
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.....	43	
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	Cancelable 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.....	58
2.9.6	Thread Read-Write Locks.....	58
2.9.7	Thread Interactions with Regular File Operations	58
2.9.8	Use of Application-Managed Thread Stacks	58
2.10	Sockets.....	59
2.10.1	Address Families.....	59
2.10.2	Addressing	59
2.10.3	Protocols	60
2.10.4	Routing.....	60
2.10.5	Interfaces.....	60
2.10.6	Socket Types.....	60
2.10.7	Socket I/O Mode.....	61
2.10.8	Socket Owner.....	61
2.10.9	Socket Queue Limits	61
2.10.10	Pending Error.....	61
2.10.11	Socket Receive Queue.....	61
2.10.12	Socket Out-of-Band Data State	62
2.10.13	Connection Indication Queue	62
2.10.14	Signals	63
2.10.15	Asynchronous Errors	63
2.10.16	Use of Options.....	64
2.10.17	Use of Sockets for Local UNIX Connections.....	67
2.10.17.1	Headers	67
2.10.18	Use of Sockets over Internet Protocols.....	67
2.10.19	Use of Sockets over Internet Protocols Based on IPv4	68
2.10.19.1	Headers	68
2.10.20	Use of Sockets over Internet Protocols Based on IPv6	68
2.10.20.1	Addressing	68
2.10.20.2	Compatibility with IPv4.....	69
2.10.20.3	Interface Identification.....	69
2.10.20.4	Options.....	70
2.10.20.5	Headers	71
2.11	Tracing.....	71
2.11.1	Tracing Data Definitions.....	72
2.11.1.1	Structures.....	72
2.11.1.2	Trace Stream Attributes.....	76
2.11.2	Trace Event Type Definitions	77
2.11.2.1	System Trace Event Type Definitions	77

Contents

2.11.2.2	User Trace Event Type Definitions.....	80
2.11.3	Trace Functions.....	80
2.12	Data Types.....	81
Chapter 3	System Interfaces	85
	Index.....	1689

List of Tables

2-1	Value of Level for Socket Options	64
2-2	Socket-Level Options	65
2-3	Trace Option: System Trace Events.....	78
2-4	Trace and Trace Event Filter Options: System Trace Events.....	79
2-5	Trace and Trace Log Options: System Trace Events	79
2-6	Trace, Trace Log, and Trace Event Filter Options: System Trace Events	80
2-7	Trace Option: User Trace Event	80

Foreword

Structure of the Standard

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

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

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

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

Introduction

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

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

The Austin Group

This standard was developed, and is maintained, by a joint working group of members of the IEEE Portable Applications Standards Committee, members of The Open Group, and members of ISO/IEC Joint Technical Committee 1. This joint working group is known as the Austin Group.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, The Open Group's Technical Standard designation, and an ISO/IEC designation. This set of specifications forms the core of the Single UNIX Specification, Version 3.

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

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

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

Background

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

Conceptually, this standard describes a set of fundamental services needed for the efficient construction of application programs. Access to these services has been provided by defining an interface, using the C programming language, a command interpreter, and common utility programs that establish standard semantics and syntax. Since this interface enables application writers to write portable applications—it was developed with that goal in mind—it has been designated POSIX,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 1003.1-2001/Cor 2-2004 was published, there were no known historical implementations that did not have to change. However, there was a broad consensus on a set of functions, types, definitions, and concepts that formed an interface that was common to most historical implementations.

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

when creating new specifications:

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

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

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

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

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

- Minimal Changes to Existing Application Code

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

This Standard

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

- Base Definitions
- Shell and Utilities
- System Interfaces (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	assert()	
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	PATH	
Error Number	[EINTR]	
Example Output	Hello, World	
Filename	/tmp	
Literal Character	'c', '\r', '\'	2
Literal String	"abcde"	2
Optional Items in Utility Syntax	[]	
Parameter	<directory pathname>	
Special Character	<newline>	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	
Utility Name	<i>awk</i>	
Utility Operand	<i>file_name</i>	
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

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

The Austin Group

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

Andrew Josey, Chair

Donald W. Cragun, Organizational Representative, IEEE PASC

Nicholas Stoughton, Organizational Representative, ISO/SC22 WG15

Mark Brown, Organizational Representative, The Open Group

Cathy Hughes, Technical Editor

Austin Group Technical Reviewers

Peter Anvin

Bouazza Bachar

Theodore P. Baker

Walter Briscoe

Mark Brown

Dave Butenhof

Geoff Clare

Donald W. Cragun

Lee Damico

Ulrich Drepper

Paul Eggert

Joanna Farley

Clive D.W. Feather

Andrew Gollan

Michael Gonzalez

Joseph M. Gwinn

Jon Hitchcock

Yvette Ho Sang

Cathy Hughes

Lowell G. Johnson

Andrew Josey

Michael Kavanaugh

David Korn

Marc Aurele La France

Jim Meyering

Gary Miller

Finnbarr P. Murphy

Joseph S. Myers

Sandra O'Donnell

Frank Prindle

Curtis Royster Jr.

Glen Seeds

Keld Jorn Simonsen

Raja Srinivasan

Nicholas Stoughton

Donn S. Terry

Fred Tydeman

Peter Van Der Veen

James Youngman

Jim Zepeda

Jason Zions

Austin Group Working Group Members

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

Participants

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.

Participants

IEEE-SA Standards Board

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

Donald N. Heirman, Chair
James T. Carlo, Vice-Chair
Judith Gorman, Secretary

Satish K. Aggarwal
Mark D. Bowman
Gary R. Engmann
Harold E. Epstein
H. Landis Floyd
Jay Forster*
Howard M. Frazier
Ruben D. Garzon

James H. Gurney
Richard J. Holleman
Lowell G. Johnson
Robert J. Kennelly
Joseph L. Koepfinger*
Peter H. Lips
L. Bruce McClung
Daleep C. Mohla

James W. Moore
Robert F. Munzner
Ronald C. Petersen
Gerald H. Peterson
John B. Posey
Gary S. Robinson
Akio Tojo
Donald W. Zipse

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

Alan Cookson, NIST Representative
Donald R. Volzka, TAB Representative
Yvette Ho Sang, Don Messina, Savoula Amanatidis, IEEE Project Editors

* Member Emeritus

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

The Austin Group

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

Andrew Josey, Chair

Donald W. Cragun, Organizational Representative, IEEE PASC

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

Mark Brown, Organizational Representative, The Open Group

Cathy Fox, Technical Editor

Austin Group Technical Reviewers

Theodore P. Baker

Julian Blake

Andries Brouwer

Mark Brown

Dave Butenhof

Geoff Clare

Donald W. Cragun

Ken Dawson

Ulrich Drepper

Larry Dwyer

Paul Eggert

Joanna Farley

Clive D.W. Feather

Cathy Fox

Mark Funkenhauser

Lois Goldthwaite

Andrew Gollan

Michael Gonzalez

Bruno Haible

Ben Harris

Jon Hitchcock

Andreas Jaeger

Andrew Josey

Jonathan Lennox

Nick Maclare

Jack McCann

Wilhelm Mueller

Joseph S. Myers

Frank Prindle

Kenneth Raeburn

Tim Robbins

Glen Seeds

Matthew Seitz

Keld Jorn Simonsen

Nicholas Stoughton

Alexander Terekhov

Donn S. Terry

Mike Wilson

Garrett A. Wollman

Mark Ziegast

Participants

Austin Group Working Group Members

Harold C. Adams	Clive D.W. Feather	Wilhelm Mueller
Alejandro Alonso	Yaacov Fenster	Finnbarr P. Murphy
Jay Ashford	Cathy Fox	Joseph S. Myers
Theodore P. Baker	Mark Funkenhauser	Alexey Neyman
David J. Blackwood	Lois Goldthwaite	Charles Ngethe
Julian Blake	Andrew Gollan	Peter Petrov
Mitchell Bonnett	Michael Gonzalez	Frank Prindle
Andries Brouwer	Karen D. Gordon	Vikram Punj
Mark Brown	Scott Gudgel	Kenneth Raeburn
Eric W. Burger	Joseph M. Gwinn	Francois Riche
Alan Burns	Steven A. Haaser	Tim Robbins
Dave Butenhof	Bruno Haible	Curtis Royster Jr.
Keith Chow	Charles E. Hammons	Diane Schleicher
Geoff Clare	Bryan Harold	Gil Shultz
Luis Cordova	Ben Harris	Stephen C. Schwarm
Donald W. Cragun	Barry Hedquist	Glen Seeds
Dragan Cvetkovic	Karl Heubaum	Matthew Seitz
Lee Damico	Jon Hitchcock	Keld Jorn Simonsen
Ken Dawson	Andreas Jaeger	Doug Stevenson
Jeroen Dekkers	Andrew Josey	Nicholas Stoughton
Juan Antonio De La Puente	Kenneth Lang	Alexander Terekhov
Steven J. Dovich	Pi-Cheng Law	Donn S. Terry
Ulrich Drepper	Jonathan Lennox	Mike Wilson
Dr. Sourav Dutta	Nick Maclarens	Garrett A. Wollman
Larry Dwyer	Roger J. Martin	Oren Yuen
Paul Eggert	Jack McCann	Mark Ziegast
Joanna Farley	George Miao	

The Open Group

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

Andrew Josey, Chair

Finnbarr P. Murphy, Vice-Chair

Mark Brown, Austin Group Liaison

Cathy Fox, Technical Editor

Base Working Group Members

Mark Brown

Dave Butenhof

Donald W. Cragun

Larry Dwyer

Ulrich Drepper

Joanna Farley

Andrew Gollan

Finnbarr P. Murphy

Frank Prindle

Andrew K. Roach

Curtis Royster Jr.

Nicholas Stoughton

Kenjiro Tsuji

Participants

IEEE

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

Portable Applications Standards Committee (PASC)

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

Balloting Committee

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

Alejandro Alonso	Michael Gonzalez	Charles Ngethe
Jay Ashford	Scott Gudgel	Peter Petrov
David J. Blackwood	Charles E. Hammons	Frank Prindle
Julian Blake	Bryan Harold	Vikram Punj
Mitchell Bonnett	Barry Hedquist	Francois Riche
Mark Brown	Karl Heubaum	Curtis Royster Jr.
Dave Butenhof	Lowell G. Johnson	Diane Schleicher
Keith Chow	Andrew Josey	Stephen C. Schwarm
Luis Cordova	Kenneth Lang	Gil Shultz
Donald W. Cragun	Pi-Cheng Law	Nicholas Stoughton
Steven J. Dovich	George Miao	Donn S. Terry
Dr. Sourav Dutta	Roger J. Martin	Oren Yuen
Yaakov Fenster	Finnbarr P. Murphy	Juan A. de la Puente

IEEE-SA Standards Board

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

James T. Carlo, Chair

James H. Gurney, Vice-Chair

Judith Gorman, Secretary

Sid Bennett

H. Stephen Berger

Clyde R. Camp

Richard DeBlasio

Harold E. Epstein

Julian Forster*

Howard M. Frazier

Toshio Fukuda

Arnold M. Greenspan

Raymond Hapeman

Donald M. Heirman

Richard H. Hulett

Lowell G. Johnson

Joseph L. Koepfinger*

Peter H. Lips

Nader Mehravari

Daleep C. Mohla

William J. Moylan

Malcolm V. Thaden

Geoffrey O. Thompson

Howard L. Wolfman

Don Wright

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

Alan Cookson, NIST Representative

Satish K. Aggarwal, NRC Representative

Savoula Amanatidis, IEEE Standards Managing Editor

* Member Emeritus

Participants

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

The Austin Group

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

Andrew Josey, Chair

Donald W. Cragun, Organizational Representative, IEEE PASC

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

Mark Brown, Organizational Representative, The Open Group

Cathy Fox, Technical Editor

Austin Group Technical Reviewers

Jay Ashford	Clive D.W. Feather	Rajesh Moorkath
Julian Blake	Yaacov Fenster	Peter Petrov
Mitchell Bonnett	Mark Funkenhauser	Franklin Prindle
Andries Brouwer	Ernesto Garcia	Vikram Punj
Mark Brown	Andrew Gollan	Eusebio Rufian-Zilberman
Paul Buerger	Michael Gonzalez	Joerg Schilling
Dave Butenhof	Jean-Denis Gorin	Stephen Schwarm
Keith Chow	Matthew Gream	Gil Shultz
Geoff Clare	Scott Gudgel	Keld Simonsen
Donald W. Cragun	Bruno Haible	Nicholas Stoughton
Lee Damico	Charles Hammons	Alexander Terekhov
Maulik Dave	Barry Hedquist	Donn Terry
Juan A. de la Puente	Jon Hitchcock	Mark-Rene Uchida
Guru Dutt Dhingra	Lowell G. Johnson	Thomas Unsicker
Loic Domaigne	Andrew Josey	Scott Valcourt
Ulrich Drepper	Piotr Karocki	Mats Wichmann
Sourav Dutta	David Leciston	Garrett A. Wollman
Larry Dwyer	Ryan Madron	Oren Yuen
Paul Eggert	Roger J. Martin	Mark Ziegast
Joanna Farley	George Miao	

Austin Group Working Group Members

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

The Open Group

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

Andrew Josey, Chair

Mark Brown, Austin Group Liaison

Cathy Fox, Technical Editor

Base Working Group Members

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

IEEE

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

Portable Applications Standards Committee (PASC)

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

Balloting Committee

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

Jay Ashford	Matthew Gream	Rajesh Moorkath
Julian Blake	Scott Gudgel	Peter Petrov
Mark Brown	Charles Hammons	Vikram Punj
Keith Chow	Barry Hedquist	Eusebio Rufian-Zilberman
Donald W. Cragun	Andrew Josey	Stephen Schwarm
Juan A. de la Puente	Piotr Karocki	Gil Shultz
Guru Dutt Dhingra	David Leciston	Mark-Rene Uchida
Ernesto Garcia	Ryan Madron	Thomas Unsicker
Michael Gonzalez	Roger J. Martin	Scott Valcourt
Jean-Denis Gorin	George Miao	Oren Yuen

Participants

IEEE-SA Standards Board

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

Don Wright, Chair
Judith Gorman, Secretary

Chuck Adams
H. Stephen Berger
Mark D. Bowman
Joseph A. Bruder
Bob Davis
Roberto de Boisson
Julian Forster*
Arnold M. Greenspan
Mark S. Halpin

Raymond Hapeman
Richard J. Holleman
Richard H. Hulett
Lowell G. Johnson
Hermann Koch
Joseph L. Koepfinger*
Thomas J. McGean
Steve M. Mills
Daleep C. Mohla

Paul Nikolich
T. W. Olsen
Ronald C. Petersen
Gary S. Robinson
Frank Stone
Malcolm V. Thaden
Doug Topping
Joe D. Watson

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

Satish K. Aggarwal, NRC Representative
Richard DeBlasio, DOE Representative
Alan Cookson, NIST Representative

Savoula Amanatidis, IEEE Standards Managing Editor

* Member Emeritus

Trademarks

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

1003.1TM 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.

BSDTM 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.

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

All other trademarks are the property of their respective owners.

POSIX[®] 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 2181**
Clarifications to the DNS Specification, R. Elz, R. Bush, July 1997.
- IETF RFC 2373**
Internet Protocol, Version 6 (IPv6) Addressing Architecture, S. Deering, R. Hinden, July 1998.
- IETF RFC 2460**
Internet Protocol, Version 6 (IPv6), S. Deering, R. Hinden, December 1998.
- Internationalisation Guide**
Guide, July 1993, Internationalisation Guide, Version 2 (ISBN: 1-859120-02-4, G304), published by The Open Group.
- ISO C (1990)**
ISO/IEC 9899:1990, Programming Languages — C, including Amendment 1:1995 (E), C Integrity (Multibyte Support Extensions (MSE) for ISO C).

ISO 2375:1985

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

ISO 8652:1987

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

ISO/IEC 1539:1990

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

ISO/IEC 4873:1991

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

ISO/IEC 6429:1992

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

ISO/IEC 6937:1994

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

ISO/IEC 8802-3:1996

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

ISO/IEC 8859

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

Part 1: Latin Alphabet No. 1

Part 2: Latin Alphabet No. 2

Part 3: Latin Alphabet No. 3

Part 4: Latin Alphabet No. 4

Part 5: Latin/Cyrillic Alphabet

Part 6: Latin/Arabic Alphabet

Part 7: Latin/Greek Alphabet

Part 8: Latin/Hebrew Alphabet

Part 9: Latin Alphabet No. 5

Part 10: Latin Alphabet No. 6

Part 11: Latin/Thai Alphabet

Part 13: Latin Alphabet No. 7

Part 14: Latin Alphabet No. 8

Part 15: Latin Alphabet No. 9

Part 16: Latin Alphabet No. 10

ISO POSIX-1:1996

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

ISO POSIX-2:1993

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

Referenced Documents

by ANSI/IEEE Std 1003.2a-1992).

Issue 1

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

Issue 2

X/Open Portability Guide, January 1987:

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

Issue 3

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

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

Issue 4

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

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

Issue 4, Version 2

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

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

Issue 5

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

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

Knuth Article

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

KornShell

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

MSE Working Draft

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

POSIX.0: 1995

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

POSIX.1: 1988

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

POSIX.1: 1990

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

POSIX.1a

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

POSIX.1d: 1999

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

POSIX.1g: 2000

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

POSIX.1j: 2000

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

POSIX.1q: 2000

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

POSIX.2b

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

POSIX.2d: 1994

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

Referenced Documents

POSIX.13:-1998

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

Sarwate Article

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

Sprung, Sha, and Lehoczky

Sprung, 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 Courses, Issue 4, Version 2

CAE Specification, May 1996, X/Open Courses, 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

2 The scope of IEEE Std 1003.1-2001 is described in the Base Definitions volume of
3 IEEE Std 1003.1-2001.
4

1.2 Conformance

5 Conformance requirements for IEEE Std 1003.1-2001 are defined in the Base Definitions volume
6 of IEEE Std 1003.1-2001, Chapter 2, Conformance.
7

1.3 Normative References

8 Normative references for IEEE Std 1003.1-2001 are defined in the Base Definitions volume of
9 IEEE Std 1003.1-2001.
10

1.4 Change History

11 Change history is described in the Rationale (Informative) volume of IEEE Std 1003.1-2001, and
12 in the CHANGE HISTORY section of reference pages.
13

1.5 Terminology

14 This section appears in the Base Definitions volume of IEEE Std 1003.1-2001, but is repeated here
15 for convenience:
16

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

18 **can**

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

22 **implementation-defined**

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

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

30 **legacy**

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

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

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

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

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

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

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

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

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

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

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

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

67 **1.6 Definitions**

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

69 **1.7 Relationship to Other Formal Standards**

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

72 ISO C (1999)

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

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

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

82 **1.8 Portability**

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

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

90 **1.8.1 Codes**

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

93 ADV **Advisory Information**

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

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

99 AIO **Asynchronous Input and Output**

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

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

105 BAR **Barriers**

106 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 FORTRAN 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 FORTRAN 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 legend.
153	FSC	File Synchronization
154		The functionality described is optional. The functionality described is also an extension to the 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 margin legend.
159	IP6	IPV6
160		The functionality described is optional. The functionality described is also an extension to the 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 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 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 margin legend.
172		Refer to the Base Definitions volume of IEEE Std 1003.1-2001, Section 1.5.2, Margin Code 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 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 margin legend.
181		Refer to the Base Definitions volume of IEEE Std 1003.1-2001, Section 1.5.2, Margin Code Notation.
183	MC3	Memory Mapped Files, Shared Memory Objects, or Typed Memory Objects
184		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
186		This is a shorthand notation for combinations of multiple option codes.
187		Where applicable, functions are marked with the MC3 margin legend in the SYNOPSIS section.
188		Where additional semantics apply to a function, the material is identified by use of the MC3 margin legend.
190		Refer to the Base Definitions volume of IEEE Std 1003.1-2001, Section 1.5.2, Margin Code Notation.
192	MF	Memory Mapped Files
193		The functionality described is optional. The functionality described is also an extension to the

194	ISO C standard.
195	Where applicable, functions are marked with the MF margin legend in the SYNOPSIS section.
196	Where additional semantics apply to a function, the material is identified by use of the MF margin legend.
197	
198 ML	Process Memory Locking
199	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
200	
201	Where applicable, functions are marked with the ML margin legend in the SYNOPSIS section.
202	Where additional semantics apply to a function, the material is identified by use of the ML margin legend.
203	
204 MLR	Range Memory Locking
205	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
206	
207	Where applicable, functions are marked with the MLR margin legend in the SYNOPSIS section.
208	Where additional semantics apply to a function, the material is identified by use of the MLR margin legend.
209	
210 MON	Monotonic Clock
211	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
212	
213	Where applicable, functions are marked with the MON margin legend in the SYNOPSIS section.
214	Where additional semantics apply to a function, the material is identified by use of the MON margin legend.
215	
216 MPR	Memory Protection
217	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
218	
219	Where applicable, functions are marked with the MPR margin legend in the SYNOPSIS section.
220	Where additional semantics apply to a function, the material is identified by use of the MPR margin legend.
221	
222 MSG	Message Passing
223	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
224	
225	Where applicable, functions are marked with the MSG margin legend in the SYNOPSIS section.
226	Where additional semantics apply to a function, the material is identified by use of the MSG margin legend.
227	
228 MX	IEC 60559 Floating-Point Option
229	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
230	
231	Where applicable, functions are marked with the MX margin legend in the SYNOPSIS section.
232	Where additional semantics apply to a function, the material is identified by use of the MX margin legend.
233	
234 OB	Obsolescent
235	The functionality described may be withdrawn in a future version of this volume of IEEE Std 1003.1-2001. Strictly Conforming POSIX Applications and Strictly Conforming XSI Applications shall not use obsolescent features.
236	
237	

238		Where applicable, the material is identified by use of the OB margin legend.
239	OF	Output Format Incompletely Specified The functionality described is an XSI extension. The format of the output produced by the utility is not fully specified. It is therefore not possible to post-process this output in a consistent fashion. Typical problems include unknown length of strings and unspecified field delimiters.
240		
241		
242		
243		Where applicable, the material is identified by use of the OF margin legend.
244	OH	Optional Header In the SYNOPSIS section of some interfaces in the System Interfaces volume of IEEE Std 1003.1-2001 an included header is marked as in the following example:
245		
246		
247	OH	#include <sys/types.h> #include <grp.h> struct group *getgrnam(const char *name);
248		
249		
250		The OH margin legend indicates that the marked header is not required on XSI-conformant systems.
251		
252	PIO	Prioritized Input and Output The functionality described is optional. The functionality described is also an extension to the ISO C standard.
253		
254		
255		Where applicable, functions are marked with the PIO margin legend in the SYNOPSIS section.
256		Where additional semantics apply to a function, the material is identified by use of the PIO margin legend.
257		
258	PS	Process Scheduling The functionality described is optional. The functionality described is also an extension to the ISO C standard.
259		
260		
261		Where applicable, functions are marked with the PS margin legend in the SYNOPSIS section.
262		Where additional semantics apply to a function, the material is identified by use of the PS margin legend.
263		
264	RS	Raw Sockets The functionality described is optional. The functionality described is also an extension to the ISO C standard.
265		
266		
267		Where applicable, functions are marked with the RS margin legend in the SYNOPSIS section.
268		Where additional semantics apply to a function, the material is identified by use of the RS margin legend.
269		
270	RTS	Realtime Signals Extension The functionality described is optional. The functionality described is also an extension to the ISO C standard.
271		
272		
273		Where applicable, functions are marked with the RTS margin legend in the SYNOPSIS section.
274		Where additional semantics apply to a function, the material is identified by use of the RTS margin legend.
275		
276	SD	Software Development Utilities The functionality described is optional.
277		
278		Where applicable, utilities are marked with the SD margin legend in the SYNOPSIS section.
279		Where additional semantics apply to a utility, the material is identified by use of the SD margin legend.
280		

281	SEM	Semaphores
282		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
284		Where applicable, functions are marked with the SEM margin legend in the SYNOPSIS section.
285		Where additional semantics apply to a function, the material is identified by use of the SEM margin legend.
287	SHM	Shared Memory Objects
288		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
290		Where applicable, functions are marked with the SHM margin legend in the SYNOPSIS section.
291		Where additional semantics apply to a function, the material is identified by use of the SHM margin legend.
293	SIO	Synchronized Input and Output
294		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
296		Where applicable, functions are marked with the SIO margin legend in the SYNOPSIS section.
297		Where additional semantics apply to a function, the material is identified by use of the SIO margin legend.
299	SPI	Spin Locks
300		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
302		Where applicable, functions are marked with the SPI margin legend in the SYNOPSIS section.
303		Where additional semantics apply to a function, the material is identified by use of the SPI margin legend.
305	SPN	Spawn
306		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
308		Where applicable, functions are marked with the SPN margin legend in the SYNOPSIS section.
309		Where additional semantics apply to a function, the material is identified by use of the SPN margin legend.
311	SS	Process Sporadic Server
312		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
314		Where applicable, functions are marked with the SS margin legend in the SYNOPSIS section.
315		Where additional semantics apply to a function, the material is identified by use of the SS margin legend.
317	TCT	Thread CPU-Time Clocks
318		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
320		Where applicable, functions are marked with the TCT margin legend in the SYNOPSIS section.
321		Where additional semantics apply to a function, the material is identified by use of the TCT margin legend.
323	TEF	Trace Event Filter
324		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
325		

326	Where applicable, functions are marked with the TEF margin legend in the SYNOPSIS section.
327	Where additional semantics apply to a function, the material is identified by use of the TEF margin legend.
329	Threads
330	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
332	Where applicable, functions are marked with the THR margin legend in the SYNOPSIS section.
333	Where additional semantics apply to a function, the material is identified by use of the THR margin legend.
335	Timeouts
336	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
338	Where applicable, functions are marked with the TMO margin legend in the SYNOPSIS section.
339	Where additional semantics apply to a function, the material is identified by use of the TMO margin legend.
341	Timers
342	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
344	Where applicable, functions are marked with the TMR margin legend in the SYNOPSIS section.
345	Where additional semantics apply to a function, the material is identified by use of the TMR margin legend.
347	Thread Priority Inheritance
348	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
350	Where applicable, functions are marked with the TPI margin legend in the SYNOPSIS section.
351	Where additional semantics apply to a function, the material is identified by use of the TPI margin legend.
353	Thread Priority Protection
354	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
356	Where applicable, functions are marked with the TPP margin legend in the SYNOPSIS section.
357	Where additional semantics apply to a function, the material is identified by use of the TPP margin legend.
359	Thread Execution Scheduling
360	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
362	Where applicable, functions are marked with the TPS margin legend for the SYNOPSIS section.
363	Where additional semantics apply to a function, the material is identified by use of the TPS margin legend.
365	Trace
366	The functionality described is optional. The functionality described is also an extension to the ISO C standard.
368	Where applicable, functions are marked with the TRC margin legend in the SYNOPSIS section.
369	Where additional semantics apply to a function, the material is identified by use of the TRC margin legend.
370	

371	TRI	Trace Inherit
372		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
374		Where applicable, functions are marked with the TRI margin legend in the SYNOPSIS section.
375		Where additional semantics apply to a function, the material is identified by use of the TRI margin legend.
377	TRL	Trace Log
378		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
380		Where applicable, functions are marked with the TRL margin legend in the SYNOPSIS section.
381		Where additional semantics apply to a function, the material is identified by use of the TRL margin legend.
383	TSA	Thread Stack Address Attribute
384		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
386		Where applicable, functions are marked with the TSA margin legend for the SYNOPSIS section.
387		Where additional semantics apply to a function, the material is identified by use of the TSA margin legend.
389	TSF	Thread-Safe Functions
390		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
392		Where applicable, functions are marked with the TSF margin legend in the SYNOPSIS section.
393		Where additional semantics apply to a function, the material is identified by use of the TSF margin legend.
395	TSH	Thread Process-Shared Synchronization
396		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
398		Where applicable, functions are marked with the TSH margin legend in the SYNOPSIS section.
399		Where additional semantics apply to a function, the material is identified by use of the TSH margin legend.
401	TSP	Thread Sporadic Server
402		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
404		Where applicable, functions are marked with the TSP margin legend in the SYNOPSIS section.
405		Where additional semantics apply to a function, the material is identified by use of the TSP margin legend.
407	TSS	Thread Stack Size Attribute
408		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
410		Where applicable, functions are marked with the TSS margin legend in the SYNOPSIS section.
411		Where additional semantics apply to a function, the material is identified by use of the TSS margin legend.
413	TYM	Typed Memory Objects
414		The functionality described is optional. The functionality described is also an extension to the ISO C standard.
415		

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

419 UP User Portability Utilities
420 The functionality described is optional.
421 Where applicable, utilities are marked with the UP margin legend in the SYNOPSIS section.
422 Where additional semantics apply to a utility, the material is identified by use of the UP margin
423 legend.

424 XSI Extension
425 The functionality described is an XSI extension. Functionality marked XSI is also an extension to
426 the ISO C standard. Application writers may confidently make use of an extension on all
427 systems supporting the X/Open System Interfaces Extension.
428 If an entire SYNOPSIS section is shaded and marked XSI, all the functionality described in that
429 reference page is an extension. See the Base Definitions volume of IEEE Std 1003.1-2001, Section
430 3.439, XSI.

431 XSR XSI STREAMS
432 The functionality described is optional. The functionality described is also an extension to the
433 ISO C standard.
434 Where applicable, functions are marked with the XSR margin legend in the SYNOPSIS section.
435 Where additional semantics apply to a function, the material is identified by use of the XSR
436 margin legend.

437 1.9 Format of Entries

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

440 NAME

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

442 SYNOPSIS

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

446 `#include <stdio.h>`

447 DESCRIPTION

448 This section describes the functionality of the function or header.

449 RETURN VALUE

450 This section indicates the possible return values, if any.

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

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

458

ERRORS459
460

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.

461
462

“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.

463

EXAMPLES

464

This section is informative.

465
466
467

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.

468

APPLICATION USAGE

469

This section is informative.

470
471
472

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.

473

RATIONALE

474

This section is informative.

475
476
477

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.

478

FUTURE DIRECTIONS

479

This section is informative.

480
481

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.

482

SEE ALSO

483

This section is informative.

484

This section gives references to related information.

485

CHANGE HISTORY

486

This section is informative.

487
488

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

General Information

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

492 **2.1 Use and Implementation of Functions**

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

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

516 **2.2 The Compilation Environment**

517 **2.2.1 POSIX.1 Symbols**

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

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

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

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

533 **2.2.1.1 The _POSIX_C_SOURCE Feature Test Macro**

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

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

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

544 Identifiers in IEEE Std 1003.1-2001 may only be undefined using the **#undef** directive as
 545 described in Section 2.1 (on page 13) or Section 2.2.2. These **#undef** directives shall follow all
 546 **#include** directives of any header in IEEE Std 1003.1-2001. 1

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

549 **2.2.1.2 The _XOPEN_SOURCE Feature Test Macro**

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

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

560 **2.2.2 The Name Space**

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

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

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

579 The prefixes `posix_`, `POSIX_`, and `_POSIX_` are reserved for use by IEEE Std 1003.1-2001 and
580 other POSIX standards. Implementations may add symbols to the headers shown in the
581 following table, provided the identifiers for those symbols either:

- 582 1. Begin with the corresponding reserved prefixes in the table, or
- 583 2. Have one of the corresponding complete names in the table, or
- 584 3. End in the string indicated as a reserved suffix in the table and do not use the reserved
585 prefixes `posix_`, `POSIX_`, or `_POSIX_`, as long as the reserved suffix is in that part of the
586 name considered significant by the implementation.

587 Symbols that use the reserved prefix `_POSIX_` may be made visible by implementations in any
588 header defined by IEEE Std 1003.1-2001.

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

	Header	Prefix	Suffix	Complete Name
637				
638				
639				
640	XSI <sys/uio.h>	iov_		UIO_MAXIOV
641	<sys/un.h>	sun_		
642	<sys/utsname.h>	uts_		
643	XSI <sys/wait.h>	si_, W[A-Z], P_		
644	<termios.h>	c_		
645	<time.h>	tm_		
646	TMR	clock_, timer_, it_, tv_,		
647	TMR	CLOCK_, TIMER_		
648	XSI <ucontext.h>	uc_, ss_		
649	XSI <ulimit.h>	UL_		
650	<utime.h>	utim_		
651	XSI <utmpx.h>	ut_	_LVL, _TIME, _PROCESS	
652				
653	<wchar.h>	wcs[a-z]		
654	<wctype.h>	is[a-z], to[a-z]		
655	<wordexp.h>	we_		
656	ANY header		_t	

1

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

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

	Header	Prefix	
665			
666			
667 XSI	<dlfcn.h>	RTLD_	
668	<fcntl.h>	F_, O_, S_	
669 XSI	<fntmsg.h>	MM_	
670	<fnmatch.h>	FNM_	
671 XSI	<ftw.h>	FTW	
672	<glob.h>	GLOB_	
673	<inttypes.h>	PRI[Xa-z], SCN[Xa-z]	1
674	<math.h>	FP_[A-Z]	2
675 XSI	<ndbm.h>	DBM_	
676	<net/if.h>	IF_	
677	<netinet/in.h>	IMPLINK_, IN_, INADDR_, IP_, IPPORT_, IPPROTO_, SOCK_	
678 IP6		IPV6_, IN6_	
679	<netinet/tcp.h>	TCP_	
680 XSI	<nl_types.h>	NL_	
681 XSI	<poll.h>	POLL	
682	<regex.h>	REG_	
683	<signal.h>	SA_, SIG_[0-9a-z_],	
684 XSI		BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SI_, SS_, SV_, TRAP_	
685 XSI	<stropts.h>	FLUSH[A-Z], I_, M_, MUXID_R[A-Z], S_, SND[A-Z], STR	1
686 XSI	<syslog.h>	LOG_	
687 XSI	<sys/ipc.h>	IPC_	
688 XSI	<sys/mman.h>	PROT_, MAP_, MS_	
689 XSI	<sys/msg.h>	MSG[A-Z]	
690 XSI	<sys/resource.h>	PRIO_, RLIM_, RLIMIT_, RUSAGE_	
691 XSI	<sys/sem.h>	SEM_	
692 XSI	<sys/shm.h>	SHM[A-Z], SHM_[A-Z]	
693 XSI	<sys/socket.h>	AF_, CMSG_, MSG_, PF_, SCM_, SHUT_, SO	
694	<sys/stat.h>	S_	
695 XSI	<sys/statvfs.h>	ST_	
696 XSI	<sys/time.h>	FD_, ITIMER_	
697 XSI	<sys/uio.h>	IOV_	
698 XSI	<sys/wait.h>	BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SI_, TRAP_	
699	<termios.h>	V, I, O, TC, B[0-9] (See below.)	
700	<wordexp.h>	WRDE_	

701 The following are used to reserve complete names for the <stdint.h> header:

702	INT[0-9A-Za-z_]*_MIN	1
703	INT[0-9A-Za-z_]*_MAX	1
704	INT[0-9A-Za-z_]*_C	1
705	UINT[0-9A-Za-z_]*_MIN	1
706	UINT[0-9A-Za-z_]*_MAX	1
707	UINT[0-9A-Za-z_]*_C	1

708 **Note:** The notation [0-9] indicates any digit. The notation [A-Z] indicates any uppercase letter in the
709 portable character set. The notation [0-9a-z_] indicates any digit, any lowercase letter in the
710 portable character set, or underscore.

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

712	XSI	CBAUD	EXTB	VDSUSP
713		DEFECCHO	FLUSHO	VLNEXT
714		ECHOCTL	LOBLK	VREPRINT
715		ECHOKE	PENDIN	VSTATUS
716		ECHOPRT	SWTCH	VWERASE
717		EXTA	VDISCARD	

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

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

727	_Exit	ccoshf	csqrt	fputc	lrintl	sinh
728	abort	ccoshl	csqrft	fputs	lround	sinhf
729	abs	ccosl	csqrtrt	fputwc	lroundf	sinhl
730	acos	ceil	ctan	fputws	lroundl	sinl
731	acosf	ceilf	ctanf	fread	malloc	sprintf
732	acosh	ceilf	ctanl	free	mblen	sqrt
733	acoshf	ceill	ctgamma	freopen	mbrlen	sqrtf
734	acoshl	ceill	ctgammaf	frexp	mbrtowc	sqrtl
735	acosl	cerf	ctgammal	frexpf	mbsinit	srand
736	acosl	cerfc	ltime	frexpl	mbsrtowcs	sscanf
737	asctime	cerfcf	difftime	fscanf	mbstowcs	str[a-z]*
738	asin	cerfcf	div	fseek	mbtowc	strtof
739	asinf	erfcf	erfcf	fsetpos	mem[a-z]*	strtoimax
740	asinh	cerfl	erfcf	ftell	mktime	strtold
741	asinhf	cexmp1	erff	fwide	modf	strtoll
742	asinhl	cexmp1f	erfl	fwprintf	modff	strtoull
743	asinl	cexmp1l	errno	fwrite	modfl	strtoumax
744	asinl	cexp	exit	fwscanf	nan	swprintf
745	atan	cexp2	exp	getc	nanf	swscanf
746	atan2	cexp2f	exp2	getchar	nanl	system
747	atan2f	cexp2l	exp2f	getenv	nearbyint	tan
748	atan2l	cexpf	exp2l	gets	nearbyintf	tanf
749	atanf	cexpl	expf	getwc	nearbyintl	tanh
750	atanf	cimag	expl	getwchar	nextafterf	tanhf
751	atanh	cimagf	expm1	gmtime	nextafterl	tanhl
752	atanh	cimagnl	expm1f	hypotf	nexttoward	tanl
753	atanhf	clearerr	expm1l	hypotl	nexttowardf	tgamma
754	atanhl	clgamma	fabs	ilogb	nexttowardl	tgammaf
755	atanl	clgammaf	fabsf	ilogbf	perror	tgammal
756	atanl	clgammal	fabsl	ilogbl	pow	time

757	atexit	clock	fclose	imaxabs	powf	tmpfile
758	atof	clog	fdim	imaxdiv	powl	tmpnam
759	atoi	clog10	fdimf	is[a-z]*	printf	to[a-z]*
760	atol	clog10f	fdiml	isblank	putc	trunc
761	atoll	clog10l	feclearexcept	iswblank	putchar	truncf
762	bsearch	clog1p	fegetenv	labs	puts	truncl
763	cabs	clog1pf	fegetexceptflag	ldexp	putwc	ungetc
764	cabsf	clog1pl	fegetround	ldexpf	putwchar	ungetwc
765	cabsl	clog2	feholdexcept	ldexpl	qsort	va_end
766	cacos	clog2f	feof	ldiv	raise	vfprintf
767	cacosf	clog2l	feraiseexcept	ldiv	rand	vfscanf
768	cacosh	clogf	ferror	lgammaf	realloc	vfwprintf
769	cacoshf	clogl	fesetenv	lgammal	remainderf	vfwscanf
770	cacoshl	conj	fesetexceptflag	llabs	remainderl	vprintf
771	cacosl	conjf	fesetround	llrint	remove	vscanf
772	calloc	conjl	fetestexcept	llrintf	remquo	vsprintf
773	carg	copysign	feupdateenv	llrintl	remquof	vsscanf
774	cargf	copysignf	fflush	llround	remquol	vswprintf
775	cargl	copysignl	fgetc	llroundf	rename	vswscanf
776	casin	cos	fgetpos	llroundl	rewind	vwprintf
777	casinf	cosf	fgets	localeconv	rint	vwscanf
778	casinh	cosh	fgetwc	localtime	rintf	wcrtomb
779	casinhf	coshf	fgetws	log	rintl	wcs[a-z]*
780	casinhl	coshl	floor	log10	round	wcstof
781	casinl	cosl	floorf	log10f	roundf	wcstoimax
782	catan	cpow	floorl	log10l	roundl	wcstold
783	catanf	cpowf	fma	log1p	scalbln	wcstoll
784	catanh	cpowl	fmaf	log1pf	scalblnf	wcstoull
785	catanh	cproj	fmal	log1pl	scalblnl	wcstoumax
786	catanhf	cprojf	fmax	log2	scalbn	wctob
787	catanhf	cprojl	fmaxf	log2f	scalbnf	wctomb
788	catanhl	creal	fmaxl	log2l	scalbnl	wctrans
789	catanhl	crealf	fmin	logb	scanf	wctype
790	catanl	creall	fminf	logbf	setbuf	wcwidth
791	cbrt	csin	fminl	logbl	setjmp	wmem[a-z]*
792	cbrtf	csinf	fmod	logf	setlocale	wprintf
793	cbrtl	csinh	fmodf	logl	setvbuf	wscanf
794	ccos	csinhf	fmodl	longjmp	signal	
795	ccosf	csinhl	fopen	lrint	sin	
796	ccosh	csinl	fprintf	lrintf	sinf	

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

- 800 4. All functions and external identifiers defined in the Base Definitions volume of
 801 IEEE Std 1003.1-2001, Chapter 13, Headers are reserved for use as identifiers with external
 802 linkage.
 803 5. All the identifiers defined in this volume of IEEE Std 1003.1-2001 that have external linkage
 804 are always reserved for use as identifiers with external linkage.

805 No other identifiers are reserved.

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

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

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

2.3 Error Numbers

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

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

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

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

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

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

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

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

848 The following symbolic names identify the possible error numbers, in the context of the
849 functions specifically defined in this volume of IEEE Std 1003.1-2001; these general descriptions

850 are more precisely defined in the ERRORS sections of the functions that return them. Only these
851 symbolic names should be used in programs, since the actual value of the error number is
852 unspecified. All values listed in this section shall be unique integer constant expressions with
853 type **int** suitable for use in #if preprocessing directives, except as noted below. The values for all
854 these names shall be found in the <errno.h> header defined in the Base Definitions volume of
855 IEEE Std 1003.1-2001. The actual values are unspecified by this volume of IEEE Std 1003.1-2001.

856 [E2BIG]

857 Argument list too long. The sum of the number of bytes used by the new process image's
858 argument list and environment list is greater than the system-imposed limit of {ARG_MAX}
859 bytes.

860 or:

861 Lack of space in an output buffer.

862 or:

863 Argument is greater than the system-imposed maximum.

864 [EACCES]

865 Permission denied. An attempt was made to access a file in a way forbidden by its file
866 access permissions.

867 [EADDRINUSE]

868 Address in use. The specified address is in use.

869 [EADDRNOTAVAIL]

870 Address not available. The specified address is not available from the local system.

871 [EAFNOSUPPORT]

872 Address family not supported. The implementation does not support the specified address
873 family, or the specified address is not a valid address for the address family of the specified
874 socket.

875 [EAGAIN]

876 Resource temporarily unavailable. This is a temporary condition and later calls to the same
877 routine may complete normally.

878 [EALREADY]

879 Connection already in progress. A connection request is already in progress for the specified
880 socket.

881 [EBADF]

882 Bad file descriptor. A file descriptor argument is out of range, refers to no open file, or a
883 read (write) request is made to a file that is only open for writing (reading).

884 [EBADMSG]

885 XSR Bad message. During a *read()*, *getmsg()*, *getpmsg()*, or *ioctl()* I_RECVFD request to a
886 STREAMS device, a message arrived at the head of the STREAM that is inappropriate for
887 the function receiving the message.

888 *read()* Message waiting to be read on a STREAM is not a data message.

889 *getmsg()* or *getpmsg()* A file descriptor was received instead of a control message.

890 *ioctl()* Control or data information was received instead of a file descriptor when
891 I_RECVFD was specified.

893 or:

894 Bad Message. The implementation has detected a corrupted message.

895 [EBUSY]
896 Resource busy. An attempt was made to make use of a system resource that is not currently
897 available, as it is being used by another process in a manner that would have conflicted with
898 the request being made by this process.

899 [ECANCELED]
900 Operation canceled. The associated asynchronous operation was canceled before
901 completion.

902 [ECHILD]
903 No child process. A *wait()* or *waitpid()* function was executed by a process that had no
904 existing or unwaited-for child process.

905 [ECONNABORTED]
906 Connection aborted. The connection has been aborted.

907 [ECONNREFUSED]
908 Connection refused. An attempt to connect to a socket was refused because there was no
909 process listening or because the queue of connection requests was full and the underlying
910 protocol does not support retransmissions.

911 [ECONNRESET]
912 Connection reset. The connection was forcibly closed by the peer.

913 [EDEADLK]
914 Resource deadlock would occur. An attempt was made to lock a system resource that
915 would have resulted in a deadlock situation.

916 [EDESTADDRREQ]
917 Destination address required. No bind address was established.

918 [EDOM]
919 Domain error. An input argument is outside the defined domain of the mathematical
920 function (defined in the ISO C standard).

921 [EDQUOT]
922 Reserved.

923 [EEXIST]
924 File exists. An existing file was mentioned in an inappropriate context; for example, as a
925 new link name in the *link()* function.

926 [EFAULT]
927 Bad address. The system detected an invalid address in attempting to use an argument of a
928 call. The reliable detection of this error cannot be guaranteed, and when not detected may
929 result in the generation of a signal, indicating an address violation, which is sent to the
930 process.

931 [EFBIG]
932 File too large. The size of a file would exceed the maximum file size of an implementation or
933 offset maximum established in the corresponding file description.

934 [EHOSTUNREACH]
935 Host is unreachable. The destination host cannot be reached (probably because the host is
936 down or a remote router cannot reach it).

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

981	[EMULTIHOP]
982	Reserved.
983	[ENAMETOOLONG]
984	Filename too long. The length of a pathname exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX}. This error may also occur when pathname substitution, as a result of encountering a symbolic link during pathname resolution, results in a pathname string the size of which exceeds {PATH_MAX}.
988	[ENETDOWN]
989	Network is down. The local network interface used to reach the destination is down.
990	[ENETRESET]
991	The connection was aborted by the network.
992	[ENETUNREACH]
993	Network unreachable. No route to the network is present.
994	[ENFILE]
995	Too many files open in system. Too many files are currently open in the system. The system has reached its predefined limit for simultaneously open files and temporarily cannot accept requests to open another one.
998	[ENOBUFS]
999	No buffer space available. Insufficient buffer resources were available in the system to perform the socket operation.
1000	
1001 XSR	[ENODATA]
1002	No message available. No message is available on the STREAM head read queue.
1003	[ENODEV]
1004	No such device. An attempt was made to apply an inappropriate function to a device; for example, trying to read a write-only device such as a printer.
1005	
1006	[ENOENT]
1007	No such file or directory. A component of a specified pathname does not exist, or the pathname is an empty string.
1008	
1009	[ENOEXEC]
1010	Executable file format error. A request is made to execute a file that, although it has the appropriate permissions, is not in the format required by the implementation for executable files.
1011	
1012	
1013	[ENOLCK]
1014	No locks available. A system-imposed limit on the number of simultaneous file and record locks has been reached and no more are currently available.
1015	
1016	[ENOLINK]
1017	Reserved.
1018	[ENOMEM]
1019	Not enough space. The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.
1020	
1021	[ENOMSG]
1022	No message of the desired type. The message queue does not contain a message of the required type during XSI interprocess communication.
1023	
1024	[ENOPROTOOPT]
1025	Protocol not available. The protocol option specified to <i>setsockopt()</i> is not supported by the

1026	implementation.
1027	[ENOSPC] No space left on a device. During the <code>write()</code> function on a regular file or when extending a directory, there is no free space left on the device.
1028	
1029	
1030 XSR	[ENOSR] No STREAM resources. Insufficient STREAMS memory resources are available to perform a STREAMS-related function. This is a temporary condition; it may be recovered from if other processes release resources.
1031	
1032	
1033	
1034 XSR	[ENOSTR] Not a STREAM. A STREAM function was attempted on a file descriptor that was not associated with a STREAMS device.
1035	
1036	
1037	[ENOSYS] Function not implemented. An attempt was made to use a function that is not available in this implementation.
1038	
1039	
1040	[ENOTCONN] Socket not connected. The socket is not connected.
1041	
1042	[ENOTDIR] Not a directory. A component of the specified pathname exists, but it is not a directory, when a directory was expected.
1043	
1044	
1045	[ENOTEMPTY] Directory not empty. A directory other than an empty directory was supplied when an empty directory was expected.
1046	
1047	
1048	[ENOTSOCK] Not a socket. The file descriptor does not refer to a socket.
1049	
1050	[ENOTSUP] Not supported. The implementation does not support this feature of the Realtime Option Group.
1051	
1052	
1053	[ENOTTY] Inappropriate I/O control operation. A control function has been attempted for a file or special file for which the operation is inappropriate.
1054	
1055	
1056	[ENXIO] No such device or address. Input or output on a special file refers to a device that does not exist, or makes a request beyond the capabilities of the device. It may also occur when, for example, a tape drive is not on-line.
1057	
1058	
1059	
1060	[EOPNOTSUPP] Operation not supported on socket. The type of socket (address family or protocol) does not support the requested operation.
1061	
1062	
1063	[EOVERFLOW] Value too large to be stored in data type. An operation was attempted which would generate a value that is outside the range of values that can be represented in the relevant data type or that are allowed for a given data item.
1064	
1065	
1066	
1067	[EPERM] Operation not permitted. An attempt was made to perform an operation limited to processes with appropriate privileges or to the owner of a file or other resource.
1068	
1069	

1070	[EPIPE] Broken pipe. A write was attempted on a socket, pipe, or FIFO for which there is no process to read the data.
1073	[EPROTO] Protocol error. Some protocol error occurred. This error is device-specific, but is generally not related to a hardware failure.
1076	[EPROTONOSUPPORT] Protocol not supported. The protocol is not supported by the address family, or the protocol is not supported by the implementation.
1079	[EPROTOTYPE] Protocol wrong type for socket. The socket type is not supported by the protocol.
1081	[ERANGE] Result too large or too small. The result of the function is too large (overflow) or too small (underflow) to be represented in the available space (defined in the ISO C standard).
1084	[EROFS] Read-only file system. An attempt was made to modify a file or directory on a file system that is read-only.
1087	[ESPIPE] Invalid seek. An attempt was made to access the file offset associated with a pipe or FIFO.
1089	[ESRCH] No such process. No process can be found corresponding to that specified by the given process ID.
1092	[ESTALE] Reserved.
1094 XSR	[ETIME] STREAM <i>ioctl()</i> timeout. The timer set for a STREAMS <i>ioctl()</i> call has expired. The cause of this error is device-specific and could indicate either a hardware or software failure, or a timeout value that is too short for the specific operation. The status of the <i>ioctl()</i> operation is unspecified.
1099	[ETIMEDOUT] Connection timed out. The connection to a remote machine has timed out. If the connection timed out during execution of the function that reported this error (as opposed to timing out prior to the function being called), it is unspecified whether the function has completed some or all of the documented behavior associated with a successful completion of the function. or: Operation timed out. The time limit associated with the operation was exceeded before the operation completed.
1108	[ETXTBSY] Text file busy. An attempt was made to execute a pure-procedure program that is currently open for writing, or an attempt has been made to open for writing a pure-procedure program that is being executed.
1112	[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

1115 suspend execution.
1116 A conforming implementation may assign the same values for [EWOULDBLOCK] and
1117 [EAGAIN].
1118 [EXDEV]
1119 Improper link. A link to a file on another file system was attempted.

1120 **2.3.1 Additional Error Numbers**

1121 Additional implementation-defined error numbers may be defined in <errno.h>.

1122 **2.4 Signal Concepts**

1123 **2.4.1 Signal Generation and Delivery**

1124 A signal is said to be “generated” for (or sent to) a process or thread when the event that causes
1125 the signal first occurs. Examples of such events include detection of hardware faults, timer
1126 expiration, signals generated via the **sigevent** structure and terminal activity, as well as
1127 invocations of the *kill()* and *sigqueue()* functions. In some circumstances, the same event
1128 generates signals for multiple processes.

1129 At the time of generation, a determination shall be made whether the signal has been generated
1130 for the process or for a specific thread within the process. Signals which are generated by some
1131 action attributable to a particular thread, such as a hardware fault, shall be generated for the
1132 thread that caused the signal to be generated. Signals that are generated in association with a
1133 process ID or process group ID or an asynchronous event, such as terminal activity, shall be
1134 generated for the process.

1135 Each process has an action to be taken in response to each signal defined by the system (see
1136 Section 2.4.3 (on page 30)). A signal is said to be “delivered” to a process when the appropriate
1137 action for the process and signal is taken. A signal is said to be “accepted” by a process when the
1138 signal is selected and returned by one of the *sigwait()* functions.

1139 During the time between the generation of a signal and its delivery or acceptance, the signal is
1140 said to be “pending”. Ordinarily, this interval cannot be detected by an application. However, a
1141 signal can be “blocked” from delivery to a thread. If the action associated with a blocked signal
1142 is anything other than to ignore the signal, and if that signal is generated for the thread, the
1143 signal shall remain pending until it is unblocked, it is accepted when it is selected and returned
1144 by a call to the *sigwait()* function, or the action associated with it is set to ignore the signal.
1145 Signals generated for the process shall be delivered to exactly one of those threads within the
1146 process which is in a call to a *sigwait()* function selecting that signal or has not blocked delivery
1147 of the signal. If there are no threads in a call to a *sigwait()* function selecting that signal, and if all
1148 threads within the process block delivery of the signal, the signal shall remain pending on the
1149 process until a thread calls a *sigwait()* function selecting that signal, a thread unblocks delivery
1150 of the signal, or the action associated with the signal is set to ignore the signal. If the action
1151 associated with a blocked signal is to ignore the signal and if that signal is generated for the
1152 process, it is unspecified whether the signal is discarded immediately upon generation or
1153 remains pending.

1154 Each thread has a “signal mask” that defines the set of signals currently blocked from delivery
1155 to it. The signal mask for a thread shall be initialized from that of its parent or creating thread,
1156 or from the corresponding thread in the parent process if the thread was created as the result of a
1157 call to *fork()*. The *pthread_sigmask()*, *sigaction()*, *sigprocmask()*, and *sigsuspend()* functions control
1158 the manipulation of the signal mask.

1159 The determination of which action is taken in response to a signal is made at the time the signal
 1160 is delivered, allowing for any changes since the time of generation. This determination is
 1161 independent of the means by which the signal was originally generated. If a subsequent
 1162 occurrence of a pending signal is generated, it is implementation-defined as to whether the
 1163 signal is delivered or accepted more than once in circumstances other than those in which
 1164 queuing is required under the Realtime Signals Extension option. The order in which multiple,
 1165 simultaneously pending signals outside the range SIGRTMIN to SIGRTMAX are delivered to or
 1166 accepted by a process is unspecified.

1167 When any stop signal (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is generated for a process, any
 1168 pending SIGCONT signals for that process shall be discarded. Conversely, when SIGCONT is
 1169 generated for a process, all pending stop signals for that process shall be discarded. When
 1170 SIGCONT is generated for a process that is stopped, the process shall be continued, even if the
 1171 SIGCONT signal is blocked or ignored. If SIGCONT is blocked and not ignored, it shall remain
 1172 pending until it is either unblocked or a stop signal is generated for the process.

1173 An implementation shall document any condition not specified by this volume of
 1174 IEEE Std 1003.1-2001 under which the implementation generates signals.

1175 2.4.2 Realtime Signal Generation and Delivery

1176 RTS This section describes extensions to support realtime signal generation and delivery. This
 1177 functionality is dependent on support of the Realtime Signals Extension option (and the rest of
 1178 this section is not further shaded for this option).

1179 Some signal-generating functions, such as high-resolution timer expiration, asynchronous I/O
 1180 completion, interprocess message arrival, and the *sigqueue()* function, support the specification
 1181 of an application-defined value, either explicitly as a parameter to the function or in a **sigevent**
 1182 structure parameter. The **sigevent** structure is defined in <signal.h> and contains at least the
 1183 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 sigval)	<i>sigev_notify_function</i>	Notification function.
(pthread_attr_t*)	<i>sigev_notify_attributes</i>	Notification attributes.

1191 The *sigev_notify* member specifies the notification mechanism to use when an asynchronous
 1192 event occurs. This volume of IEEE Std 1003.1-2001 defines the following values for the
 1193 *sigev_notify* member:

- | | |
|-------------------|---|
| 1194 SIGEV_NONE | No asynchronous notification shall be delivered when the event of
1195 interest occurs. |
| 1196 SIGEV_SIGNAL | The signal specified in <i>sigev_signo</i> shall be generated for the process when
1197 the event of interest occurs. If the implementation supports the Realtime
1198 Signals Extension option and if the SA_SIGINFO flag is set for that signal
1199 number, then the signal shall be queued to the process and the value
1200 specified in <i>sigev_value</i> shall be the <i>si_value</i> component of the generated
1201 signal. If SA_SIGINFO is not set for that signal number, it is unspecified
1202 whether the signal is queued and what value, if any, is sent. |
| 1203 SIGEV_THREAD | A notification function shall be called to perform notification. |

1204 An implementation may define additional notification mechanisms.

1205 The *sigev_signo* member specifies the signal to be generated. The *sigev_value* member is the
 1206 application-defined value to be passed to the signal-catching function at the time of the signal
 1207 delivery or to be returned at signal acceptance as the *si_value* member of the **siginfo_t** structure.

1208 The **sigval** 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.

1213 The *sival_int* member shall be used when the application-defined value is of type **int**; the
 1214 *sival_ptr* member shall be used when the application-defined value is a pointer.

1215 When a signal is generated by the *sigqueue()* function or any signal-generating function that
 1216 supports the specification of an application-defined value, the signal shall be marked pending
 1217 and, if the SA_SIGINFO flag is set for that signal, the signal shall be queued to the process along
 1218 with the application-specified signal value. Multiple occurrences of signals so generated are
 1219 queued in FIFO order. It is unspecified whether signals so generated are queued when the
 1220 SA_SIGINFO flag is not set for that signal.

1221 Signals generated by the *kill()* function or other events that cause signals to occur, such as
 1222 detection of hardware faults, *alarm()* timer expiration, or terminal activity, and for which the
 1223 implementation does not support queuing, shall have no effect on signals already queued for the
 1224 same signal number.

1225 When multiple unblocked signals, all in the range SIGRTMIN to SIGRTMAX, are pending, the
 1226 behavior shall be as if the implementation delivers the pending unblocked signal with the lowest
 1227 signal number within that range. No other ordering of signal delivery is specified.

1228 If, when a pending signal is delivered, there are additional signals queued to that signal number,
 1229 the signal shall remain pending. Otherwise, the pending indication shall be reset.

1230 Multi-threaded programs can use an alternate event notification mechanism. When a
 1231 notification is processed, and the *sigev_notify* member of the **sigevent** structure has the value
 1232 SIGEV_THREAD, the function *sigev_notify_function* is called with parameter *sigev_value*.

1233 The function shall be executed in an environment as if it were the *start_routine* for a newly
 1234 created thread with thread attributes specified by *sigev_notify_attributes*. If *sigev_notify_attributes*
 1235 is NULL, the behavior shall be as if the thread were created with the *detachstate* attribute set to
 1236 PTHREAD_CREATE_DETACHED. Supplying an attributes structure with a *detachstate* attribute
 1237 of PTHREAD_CREATE_JOINABLE results in undefined behavior. The signal mask of this
 1238 thread is implementation-defined.

1239 2.4.3 Signal Actions

1240 There are three types of action that can be associated with a signal: SIG_DFL, SIG_IGN, or a
 1241 pointer to a function. Initially, all signals shall be set to SIG_DFL or SIG_IGN prior to entry of
 1242 the *main()* routine (see the *exec* functions). The actions prescribed by these values are as follows:

1243 **SIG_DFL** Signal-specific default action.

1244 The default actions for the signals defined in this volume of IEEE Std 1003.1-2001
 1245 RTS are specified under <signal.h>. If the Realtime Signals Extension option is
 1246 supported, the default actions for the realtime signals in the range SIGRTMIN to
 1247 SIGRTMAX shall be to terminate the process abnormally.

1248 If the default action is to stop the process, the execution of that process is
 1249 temporarily suspended. When a process stops, a SIGCHLD signal shall be
 1250 generated for its parent process, unless the parent process has set the
 1251 SA_NOCLDSTOP flag. While a process is stopped, any additional signals that are
 1252 sent to the process shall not be delivered until the process is continued, except
 1253 SIGKILL which always terminates the receiving process. A process that is a
 1254 member of an orphaned process group shall not be allowed to stop in response to
 1255 the SIGTSTP, SIGTTIN, or SIGTTOU signals. In cases where delivery of one of
 1256 these signals would stop such a process, the signal shall be discarded.

1257 Setting a signal action to SIG_DFL for a signal that is pending, and whose default
 1258 action is to ignore the signal (for example, SIGCHLD), shall cause the pending
 1259 RTS signal to be discarded, whether or not it is blocked. If the Realtime Signals
 1260 Extension option is supported, any queued values pending shall be discarded and
 1261 the resources used to queue them shall be released and returned to the system for
 1262 other use. 1

1263 The default action for SIGCONT is to resume execution at the point where the
 1264 process was stopped, after first handling any pending unblocked signals. 1

1265 XSI When a stopped process is continued, a SIGCHLD signal may be generated for its
 1266 parent process, unless the parent process has set the SA_NOCLDSTOP flag. 1

1267 SIG_IGN Ignore signal.

1268 Delivery of the signal shall have no effect on the process. The behavior of a process
 1269 RTS is undefined after it ignores a SIGFPE, SIGILL, SIGSEGV, or SIGBUS signal that
 1270 RTS was not generated by `kill()`, `sigqueue()`, or `raise()`.

1271 The system shall not allow the action for the signals SIGKILL or SIGSTOP to be set
 1272 to SIG_IGN.

1273 Setting a signal action to SIG_IGN for a signal that is pending shall cause the
 1274 pending signal to be discarded, whether or not it is blocked.

1275 If a process sets the action for the SIGCHLD signal to SIG_IGN, the behavior is
 1276 XSI unspecified, except as specified below.

1277 If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the
 1278 calling processes shall not be transformed into zombie processes when they
 1279 terminate. If the calling process subsequently waits for its children, and the process
 1280 has no unwaited-for children that were transformed into zombie processes, it shall
 1281 block until all of its children terminate, and `wait()`, `waitid()`, and `waitpid()` shall fail
 1282 and set `errno` to [ECHILD].

1283 RTS If the Realtime Signals Extension option is supported, any queued values pending
 1284 shall be discarded and the resources used to queue them shall be released and
 1285 made available to queue other signals.

1286 pointer to a function

1287 Catch signal.

1288 On delivery of the signal, the receiving process is to execute the signal-catching
 1289 function at the specified address. After returning from the signal-catching function,
 1290 the receiving process shall resume execution at the point at which it was
 1291 interrupted.

1292 If the SA_SIGINFO flag for the signal is cleared, the signal-catching function shall
 1293 be entered as a C-language function call as follows:

1294		<code>void func(int signo);</code>													
1295 XSI RTS		If the SA_SIGINFO flag for the signal is set, the signal-catching function shall be entered as a C-language function call as follows:													
1296															
1297		<code>void func(int signo, siginfo_t *info, void *context);</code>													
1298		where <i>func</i> is the specified signal-catching function, <i>signo</i> is the signal number of the signal being delivered, and <i>info</i> is a pointer to a siginfo_t structure defined in <signal.h> containing at least the following members:													
1299															
1300															
1301															
1302		<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">Member Type</th> <th style="text-align: center;">Member Name</th> <th style="text-align: center;">Description</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">int</td> <td style="text-align: center;"><i>si_signo</i></td> <td>Signal number.</td> </tr> <tr> <td style="text-align: center;">int</td> <td style="text-align: center;"><i>si_code</i></td> <td>Cause of the signal.</td> </tr> <tr> <td style="text-align: center;">union sigval</td> <td style="text-align: center;"><i>si_value</i></td> <td>Signal value.</td> </tr> </tbody> </table>	Member Type	Member Name	Description	int	<i>si_signo</i>	Signal number.	int	<i>si_code</i>	Cause of the signal.	union sigval	<i>si_value</i>	Signal value.	
Member Type	Member Name	Description													
int	<i>si_signo</i>	Signal number.													
int	<i>si_code</i>	Cause of the signal.													
union sigval	<i>si_value</i>	Signal value.													
1303															
1304															
1305															
1306		The <i>si_signo</i> member shall contain the signal number. This shall be the same as the <i>signo</i> parameter. The <i>si_code</i> member shall contain a code identifying the cause of the signal. The following values are defined for <i>si_code</i> :													
1307															
1308															
1309 RTS	SI_USER	The signal was sent by the <i>kill()</i> function. The implementation may set <i>si_code</i> to SI_USER if the signal was sent by the <i>raise()</i> or <i>abort()</i> functions or any similar functions provided as implementation extensions.													
1310															
1311															
1312															
1313 RTS	SI_QUEUE	The signal was sent by the <i>sigqueue()</i> function.													
1314 RTS	SI_TIMER	The signal was generated by the expiration of a timer set by <i>timer_settime()</i> .													
1315															
1316 RTS	SI_ASYNCIO	The signal was generated by the completion of an asynchronous I/O request.													
1317															
1318 RTS	SI_MESSAGEQUEUE	The signal was generated by the arrival of a message on an empty message queue.													
1319															
1320		If the signal was not generated by one of the functions or events listed above, the <i>si_code</i> shall be set to an implementation-defined value that is not equal to any of the values defined above.													
1321															
1322															
1323 RTS		If the Realtime Signals Extension is supported, and <i>si_code</i> is one of SI_QUEUE, SI_TIMER, SI_ASYNCIO, or SI_MESSAGEQUEUE, then <i>si_value</i> shall contain the application-specified signal value. Otherwise, the contents of <i>si_value</i> are undefined.													
1324															
1325															
1326															
1327 XSI		The behavior of a process is undefined after it returns normally from a signal-catching function for a SIGBUS, SIGFPE, SIGILL, or SIGSEGV signal that was not generated by <i>kill()</i> , <i>sigqueue()</i> , or <i>raise()</i> .													
1328 RTS															
1329															
1330		The system shall not allow a process to catch the signals SIGKILL and SIGSTOP.													
1331															
1332															
1333															
1334		If a process establishes a signal-catching function for the SIGCHLD signal while it has a terminated child process for which it has not waited, it is unspecified whether a SIGCHLD signal is generated to indicate that child process.													
1335															
1336															
1337															
1338															
1339															
1340															
1341															
1342															
1343															
1344															
1345															
1346															
1347															
1348															
1349															
1350															
1351															
1352															
1353															
1354															
1355															
1356															
1357															
1358															
1359															
1360															
1361															
1362															
1363															
1364															
1365															
1366															
1367															
1368															
1369															
1370															
1371															
1372															
1373															
1374															
1375															
1376															
1377															
1378															
1379															
1380															
1381															
1382															
1383															
1384															
1385															
1386															
1387															
1388															
1389															
1390															
1391															
1392															
1393															
1394															
1395															
1396															
1397															
1398															
1399															
1400															
1401															
1402															
1403															
1404															
1405															
1406															
1407															
1408															
1409															
1410															
1411															
1412															
1413															
1414															
1415															
1416															
1417															
1418															
1419															
1420															
1421															
1422															
1423															
1424															
1425															
1426															
1427															
1428															
1429															
1430															
1431															
1432															
1433															
1434															
1435															
1436															
1437															
1438															
1439															
1440															
1441															
1442															
1443															
1444															
1445															
1446															
1447															
1448															
1449															
1450															
1451															
1452															
1453															
1454															
1455															
1456															
1457															
1458															
1459															
1460															
1461															
1462															
1463															
1464															
1465															
1466															
1467															
1468															
1469															
1470															
1471															
1472															
1473															
1474															
1475															
1476															
1477															
1478															
1479															
1480															
1481															
1482															
1483															
1484															
1485															
1486															
1487															
1488															
1489															
1490															
1491															
1492															
1493															
1494															
1495															
1496															
1497															
1498															
1499															
1500															
1501															
1502															
1503															
1504															
1505															
1506															
1507															
1508															
1509															
1510															
1511															
1512															
1513															
1514															
1515															
1516															
1517															
1518															
1519															
1520															
1521															
1522															
1523															
1524															
1525															
1526															
1527															
1528															
1529															
1530															
1531															
1532															
1533															
1534															
1535															
1536															
1537															
1538															
1539															
1540															
1541															
1542															
1543															
1544															
1545															
1546															
1547															
1548															
1549															
1550															
1551															
1552															
1553															
1554															
1555															
1556															
1557															
1558															
1559															
1560															
1561															
1562															
1563															
1564															
1565															
1566															
1567															
1568															
1569															
1570															
1571															
1572															
1573															
1574															
1575															
1576															
1577															
1578															
1579															
1580															
1581															
1582															
1583															
1584															
1585															
1586															
1587															
1588															
1589															
1590															
1591															
1592															
1593															
1594															
1595															
1596															
1597															
1598															
1599															
1600															
1601															
1602															
1603															
1604															
1605															
1606															
1607															
1608															
1609															
1610															
1611															
1612															
1613															
1614															
1615															
1616															
1617															
1618															
1619															
1620															
1621															
1622															
1623															
1624															
1625															
1626															
1627															
1628															
1629															
1630															
1631															
1632															
1633															
1634															
1635															
1636															
1637															
1638															
1639															
1640															
1641															
1642															
1643															
1644															
1645															
1646															
1647															
1648															
1649															
1650															
1651															
1652															
1653															
1654															
1655															
1656															
1657															
1658															
1659															
1660															
1661															
1662															
1663															
1664															
1665															
1666															
1667															
1668															
1669															
1670															
1671															
1672															
1673															
1674															
1675															
1676															
1677															
1678															
1679															
1680															
1681															
1682															
1683															
1684															
1685															
1686															
1687															
1688															
1689															
1690															
1691															
1692															
1693															
1694															
1695															
1696															
1697															
1698															
1699															
1700															
1701															
1702															
1703															
1704															

1338 The following table defines a set of functions that shall be either reentrant or non-
 1339 interruptible by signals and shall be async-signal-safe. Therefore applications may
 1340 invoke them, without restriction, from signal-catching functions:

1341 <i>_Exit()</i>	1341 <i>fpathconf()</i>	1341 <i>read()</i>	1341 <i>sigset()</i>
1342 <i>_exit()</i>	1342 <i>fstat()</i>	1342 <i>readlink()</i>	1342 <i>sigsuspend()</i>
1343 <i>abort()</i>	1343 <i>fsync()</i>	1343 <i>recv()</i>	1343 <i>sockatmark()</i>
1344 <i>accept()</i>	1344 <i>ftruncate()</i>	1344 <i>recvfrom()</i>	1344 <i>socket()</i>
1345 <i>access()</i>	1345 <i>getegid()</i>	1345 <i>recvmsg()</i>	1345 <i>socketpair()</i>
1346 <i>aio_error()</i>	1346 <i>geteuid()</i>	1346 <i>rename()</i>	1346 <i>stat()</i>
1347 <i>aio_return()</i>	1347 <i>getgid()</i>	1347 <i>rmdir()</i>	1347 <i>symlink()</i>
1348 <i>aio_suspend()</i>	1348 <i>getgroups()</i>	1348 <i>select()</i>	1348 <i>sysconf()</i>
1349 <i>alarm()</i>	1349 <i>getpeername()</i>	1349 <i>sem_post()</i>	1349 <i>tcdrain()</i>
1350 <i>bind()</i>	1350 <i>getpgroup()</i>	1350 <i>send()</i>	1350 <i>tcflow()</i>
1351 <i>cgetattrspeed()</i>	1351 <i>getpid()</i>	1351 <i>sendmsg()</i>	1351 <i>tcflush()</i>
1352 <i>cgetattrspeed()</i>	1352 <i>getppid()</i>	1352 <i>sendto()</i>	1352 <i>tcgetattr()</i>
1353 <i>csetattrspeed()</i>	1353 <i>getsockname()</i>	1353 <i>setgid()</i>	1353 <i>tcgetattrgrp()</i>
1354 <i>csetattrspeed()</i>	1354 <i>getsockopt()</i>	1354 <i>setpgid()</i>	1354 <i>tcsendbreak()</i>
1355 <i>chdir()</i>	1355 <i>getuid()</i>	1355 <i>setsid()</i>	1355 <i>tcsetattr()</i>
1356 <i>chmod()</i>	1356 <i>kill()</i>	1356 <i>setssockopt()</i>	1356 <i>tcsetpgrp()</i>
1357 <i>chown()</i>	1357 <i>link()</i>	1357 <i>setuid()</i>	1357 <i>time()</i>
1358 <i>clock_gettime()</i>	1358 <i>listen()</i>	1358 <i>shutdown()</i>	1358 <i>timer_getoverrun()</i>
1359 <i>close()</i>	1359 <i>lseek()</i>	1359 <i>sigaction()</i>	1359 <i>timer_gettime()</i>
1360 <i>connect()</i>	1360 <i>lstat()</i>	1360 <i>sigaddset()</i>	1360 <i>timer_settime()</i>
1361 <i>creat()</i>	1361 <i>mkdir()</i>	1361 <i>sigdelset()</i>	1361 <i>times()</i>
1362 <i>dup()</i>	1362 <i>mkfifo()</i>	1362 <i>sigemptyset()</i>	1362 <i>umask()</i>
1363 <i>dup2()</i>	1363 <i>open()</i>	1363 <i>sigfillset()</i>	1363 <i>uname()</i>
1364 <i>execle()</i>	1364 <i>pathconf()</i>	1364 <i>sigismember()</i>	1364 <i>unlink()</i>
1365 <i>execve()</i>	1365 <i>pause()</i>	1365 <i>sleep()</i>	1365 <i>utime()</i>
1366 <i>fchmod()</i>	1366 <i>pipe()</i>	1366 <i>signal()</i>	1366 <i>wait()</i>
1367 <i>fchown()</i>	1367 <i>poll()</i>	1367 <i>sigpause()</i>	1367 <i>waitpid()</i>
1368 <i>fcntl()</i>	1368 <i>posix_trace_event()</i>	1368 <i>sigpending()</i>	1368 <i>write()</i>
1369 <i>fdatasync()</i>	1369 <i>pselect()</i>	1369 <i>sigprocmask()</i>	
1370 <i>fork()</i>	1370 <i>raise()</i>	1370 <i>sigqueue()</i>	

2

1371 All functions not in the above table are considered to be unsafe with respect to
 1372 signals. In the presence of signals, all functions defined by this volume of
 1373 IEEE Std 1003.1-2001 shall behave as defined when called from or interrupted by a
 1374 signal-catching function, with a single exception: when a signal interrupts an
 1375 unsafe function and the signal-catching function calls an unsafe function, the
 1376 behavior is undefined.

1377 When a signal is delivered to a thread, if the action of that signal specifies termination, stop, or
 1378 continue, the entire process shall be terminated, stopped, or continued, respectively.

1379 **2.4.4 Signal Effects on Other Functions**

1380 Signals affect the behavior of certain functions defined by this volume of IEEE Std 1003.1-2001 if
1381 delivered to a process while it is executing such a function. If the action of the signal is to
1382 terminate the process, the process shall be terminated and the function shall not return. If the
1383 action of the signal is to stop the process, the process shall stop until continued or terminated.
1384 Generation of a SIGCONT signal for the process shall cause the process to be continued, and the
1385 original function shall continue at the point the process was stopped. If the action of the signal is
1386 to invoke a signal-catching function, the signal-catching function shall be invoked; in this case
1387 the original function is said to be “interrupted” by the signal. If the signal-catching function
1388 executes a **return** statement, the behavior of the interrupted function shall be as described
1389 individually for that function, except as noted for unsafe functions. Signals that are ignored shall
1390 not affect the behavior of any function; signals that are blocked shall not affect the behavior of
1391 any function until they are unblocked and then delivered, except as specified for the *sigpending()*
1392 and *sigwait()* functions.

1393 **2.5 Standard I/O Streams**

1394 A stream is associated with an external file (which may be a physical device) by “opening” a file,
1395 which may involve “creating” a new file. Creating an existing file causes its former contents to
1396 be discarded if necessary. If a file can support positioning requests (such as a disk file, as
1397 opposed to a terminal), then a “file position indicator” associated with the stream is positioned
1398 at the start (byte number 0) of the file, unless the file is opened with append mode, in which case
1399 it is implementation-defined whether the file position indicator is initially positioned at the
1400 beginning or end of the file. The file position indicator is maintained by subsequent reads,
1401 writes, and positioning requests, to facilitate an orderly progression through the file. All input
1402 takes place as if bytes were read by successive calls to *fgetc()*; all output takes place as if bytes
1403 were written by successive calls to *fputc()*.

1404 When a stream is “unbuffered”, bytes are intended to appear from the source or at the
1405 destination as soon as possible; otherwise, bytes may be accumulated and transmitted as a block.
1406 When a stream is “fully buffered”, bytes are intended to be transmitted as a block when a buffer
1407 is filled. When a stream is “line buffered”, bytes are intended to be transmitted as a block when a
1408 newline byte is encountered. Furthermore, bytes are intended to be transmitted as a block when
1409 a buffer is filled, when input is requested on an unbuffered stream, or when input is requested
1410 on a line-buffered stream that requires the transmission of bytes. Support for these
1411 characteristics is implementation-defined, and may be affected via *setbuf()* and *setvbuf()*.

1412 A file may be disassociated from a controlling stream by “closing” the file. Output streams are
1413 flushed (any unwritten buffer contents are transmitted) before the stream is disassociated from
1414 the file. The value of a pointer to a **FILE** object is unspecified after the associated file is closed
1415 (including the standard streams).

1416 A file may be subsequently reopened, by the same or another program execution, and its
1417 contents reclaimed or modified (if it can be repositioned at its start). If the *main()* function
1418 returns to its original caller, or if the *exit()* function is called, all open files are closed (hence all
1419 output streams are flushed) before program termination. Other paths to program termination,
1420 such as calling *abort()*, need not close all files properly.

1421 The address of the **FILE** object used to control a stream may be significant; a copy of a **FILE**
1422 object need not necessarily serve in place of the original.

1423 At program start-up, three streams are predefined and need not be opened explicitly: *standard*
1424 *input* (for reading conventional input), *standard output* (for writing conventional output), and

1425 *standard error* (for writing diagnostic output). When opened, the standard error stream is not
 1426 fully buffered; the standard input and standard output streams are fully buffered if and only if
 1427 the stream can be determined not to refer to an interactive device.

1428 2.5.1 Interaction of File Descriptors and Standard I/O Streams

1429 CX This section describes the interaction of file descriptors and standard I/O streams. This
 1430 functionality is an extension to the ISO C standard (and the rest of this section is not further CX
 1431 shaded).

1432 An open file description may be accessed through a file descriptor, which is created using
 1433 functions such as *open()* or *pipe()*, or through a stream, which is created using functions such as
 1434 *fopen()* or *popen()*. Either a file descriptor or a stream is called a ‘‘handle’’ on the open file
 1435 description to which it refers; an open file description may have several handles.

1436 Handles can be created or destroyed by explicit user action, without affecting the underlying
 1437 open file description. Some of the ways to create them include *fcntl()*, *dup()*, *fdopen()*, *fileno()*,
 1438 and *fork()*. They can be destroyed by at least *fclose()*, *close()*, and the *exec* functions.

1439 A file descriptor that is never used in an operation that could affect the file offset (for example,
 1440 *read()*, *write()*, or *lseek()*) is not considered a handle for this discussion, but could give rise to one
 1441 (for example, as a consequence of *fdopen()*, *dup()*, or *fork()*). This exception does not include the
 1442 file descriptor underlying a stream, whether created with *fopen()* or *fdopen()*, so long as it is not
 1443 used directly by the application to affect the file offset. The *read()* and *write()* functions
 1444 implicitly affect the file offset; *lseek()* explicitly affects it.

1445 The result of function calls involving any one handle (the ‘‘active handle’’) is defined elsewhere
 1446 in this volume of IEEE Std 1003.1-2001, but if two or more handles are used, and any one of them
 1447 is a stream, the application shall ensure that their actions are coordinated as described below. If
 1448 this is not done, the result is undefined.

1449 A handle which is a stream is considered to be closed when either an *fclose()* or *freopen()* is
 1450 executed on it (the result of *freopen()* is a new stream, which cannot be a handle on the same
 1451 open file description as its previous value), or when the process owning that stream terminates
 1452 with *exit()*, *abort()*, or due to a signal. A file descriptor is closed by *close()*, *_exit()*, or the *exec*
 1453 functions when FD_CLOEXEC is set on that file descriptor.

1454 For a handle to become the active handle, the application shall ensure that the actions below are
 1455 performed between the last use of the handle (the current active handle) and the first use of the
 1456 second handle (the future active handle). The second handle then becomes the active handle. All
 1457 activity by the application affecting the file offset on the first handle shall be suspended until it
 1458 again becomes the active file handle. (If a stream function has as an underlying function one that
 1459 affects the file offset, the stream function shall be considered to affect the file offset.)

1460 The handles need not be in the same process for these rules to apply.

1461 Note that after a *fork()*, two handles exist where one existed before. The application shall ensure
 1462 that, if both handles can ever be accessed, they are both in a state where the other could become
 1463 the active handle first. The application shall prepare for a *fork()* exactly as if it were a change of
 1464 active handle. (If the only action performed by one of the processes is one of the *exec* functions or
 1465 *_exit()* (not *exit()*), the handle is never accessed in that process.)

1466 For the first handle, the first applicable condition below applies. After the actions required
 1467 below are taken, if the handle is still open, the application can close it.

- 1468 • If it is a file descriptor, no action is required.

- 1469 • If the only further action to be performed on any handle to this open file descriptor is to close
 1470 it, no action need be taken.

- 1471 • If it is a stream which is unbuffered, no action need be taken.

- 1472 • If it is a stream which is line buffered, and the last byte written to the stream was a
 1473 <newline> (that is, as if a:

1474 `putc('\n')`

1

1475 was the most recent operation on that stream), no action need be taken.

- 1476 • If it is a stream which is open for writing or appending (but not also open for reading), the
 1477 application shall either perform an *fflush()*, or the stream shall be closed.

- 1478 • If the stream is open for reading and it is at the end of the file (*feof()* is true), no action need
 1479 be taken.

- 1480 • If the stream is open with a mode that allows reading and the underlying open file
 1481 description refers to a device that is capable of seeking, the application shall either perform
 1482 an *fflush()*, or the stream shall be closed.

1483 Otherwise, the result is undefined.

1484 For the second handle:

- 1485 • If any previous active handle has been used by a function that explicitly changed the file
 1486 offset, except as required above for the first handle, the application shall perform an *lseek()* or
 1487 *fseek()* (as appropriate to the type of handle) to an appropriate location.

1488 If the active handle ceases to be accessible before the requirements on the first handle, above,
 1489 have been met, the state of the open file description becomes undefined. This might occur during
 1490 functions such as a *fork()* or *_exit()*.

1491 The *exec* functions make inaccessible all streams that are open at the time they are called,
 1492 independent of which streams or file descriptors may be available to the new process image.

1493 When these rules are followed, regardless of the sequence of handles used, implementations
 1494 shall ensure that an application, even one consisting of several processes, shall yield correct
 1495 results: no data shall be lost or duplicated when writing, and all data shall be written in order,
 1496 except as requested by seeks. It is implementation-defined whether, and under what conditions,
 1497 all input is seen exactly once.

1498 If the rules above are not followed, the result is unspecified.

1499 Each function that operates on a stream is said to have zero or more “underlying functions”.
 1500 This means that the stream function shares certain traits with the underlying functions, but does
 1501 not require that there be any relation between the implementations of the stream function and its
 1502 underlying functions.

1503 2.5.2 Stream Orientation and Encoding Rules

1504 For conformance to the ISO/IEC 9899:1999 standard, the definition of a stream includes an
 1505 “orientation”. After a stream is associated with an external file, but before any operations are
 1506 performed on it, the stream is without orientation. Once a wide-character input/output function
 1507 has been applied to a stream without orientation, the stream shall become “wide-oriented”.
 1508 Similarly, once a byte input/output function has been applied to a stream without orientation,
 1509 the stream shall become “byte-oriented”. Only a call to the *freopen()* function or the *fwipe()*
 1510 function can otherwise alter the orientation of a stream.

1511 A successful call to *freopen()* shall remove any orientation. The three predefined streams *standard input*, *standard output*, and *standard error* shall be unoriented at program start-up.

1513 Byte input/output functions cannot be applied to a wide-oriented stream, and wide-character
1514 input/output functions cannot be applied to a byte-oriented stream. The remaining stream
1515 operations shall not affect and shall not be affected by a stream's orientation, except for the
1516 following additional restriction:

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

1520 Each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state
1521 of the stream. A successful call to *fgetpos()* shall store a representation of the value of this
1522 **mbstate_t** object as part of the value of the **fpos_t** object. A later successful call to *fsetpos()* using
1523 the same stored **fpos_t** value shall restore the value of the associated **mbstate_t** object as well as
1524 the position within the controlled stream.

1525 Implementations that support multiple encoding rules associate an encoding rule with the
1526 stream. The encoding rule shall be determined by the setting of the *LC_CTYPE* category in the
1527 current locale at the time when the stream becomes wide-oriented. As with the stream's
1528 orientation, the encoding rule associated with a stream cannot be changed once it has been set,
1529 except by a successful call to *freopen()* which clears the encoding rule and resets the orientation
1530 to unoriented.

1531 Although wide-oriented streams are conceptually sequences of wide characters, the external file
1532 associated with a wide-oriented stream is a sequence of (possibly multi-byte) characters
1533 generalized as follows:

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

1537 Moreover, the encodings used for characters may differ among files. Both the nature and choice
1538 of such encodings are implementation-defined.

1539 The wide-character input functions read characters from the stream and convert them to wide
1540 characters as if they were read by successive calls to the *fgetwc()* function. Each conversion shall
1541 occur as if by a call to the *mbtowc()* function, with the conversion state described by the stream's
1542 CX own **mbstate_t** object, except the encoding rule associated with the stream is used instead of the
1543 encoding rule implied by the *LC_CTYPE* category of the current locale.

1544 The wide-character output functions convert wide characters to (possibly multi-byte) characters
1545 and write them to the stream as if they were written by successive calls to the *fputwc()* function.
1546 Each conversion shall occur as if by a call to the *wrtomb()* function, with the conversion state
1547 CX described by the stream's own **mbstate_t** object, except the encoding rule associated with the
1548 stream is used instead of the encoding rule implied by the *LC_CTYPE* category of the current
1549 locale.

1550 An “encoding error” shall occur if the character sequence presented to the underlying *mbtowc()*
1551 function does not form a valid (generalized) character, or if the code value passed to the
1552 underlying *wrtomb()* function does not correspond to a valid (generalized) character. The
1553 wide-character input/output functions and the byte input/output functions store the value of
1554 the macro [EILSEQ] in *errno* if and only if an encoding error occurs.

1555 2.6 STREAMS

1556 XSR STREAMS functionality is provided on implementations supporting the XSI STREAMS Option
1557 Group. This functionality is dependent on support of the XSI STREAMS option (and the rest of
1558 this section is not further shaded for this option).

1559 STREAMS provides a uniform mechanism for implementing networking services and other
1560 character-based I/O. The STREAMS function provides direct access to protocol modules.
1561 STREAMS modules are unspecified objects. Access to STREAMS modules is provided by
1562 interfaces in IEEE Std 1003.1-2001. Creation of STREAMS modules is outside the scope of
1563 IEEE Std 1003.1-2001.

1564 A STREAM is typically a full-duplex connection between a process and an open device or
1565 pseudo-device. However, since pipes may be STREAMS-based, a STREAM can be a full-duplex
1566 connection between two processes. The STREAM itself exists entirely within the implementation
1567 and provides a general character I/O function for processes. It optionally includes one or more
1568 intermediate processing modules that are interposed between the process end of the STREAM
1569 (STREAM head) and a device driver at the end of the STREAM (STREAM end).

1570 STREAMS I/O is based on messages. There are three types of message:

- 1571 • *Data messages* containing actual data for input or output
- 1572 • *Control data* containing instructions for the STREAMS modules and underlying
1573 implementation
- 1574 • Other messages, which include file descriptors

1575 The interface between the STREAM and the rest of the implementation is provided by a set of
1576 functions at the STREAM head. When a process calls *write()*, *writev()*, *putmsg()*, *putpmsg()*, or
1577 *ioctl()*, messages are sent down the STREAM, and *read()*, *readv()*, *getmsg()*, or *getpmsg()* accepts
1578 data from the STREAM and passes it to a process. Data intended for the device at the
1579 downstream end of the STREAM is packaged into messages and sent downstream, while data
1580 and signals from the device are composed into messages by the device driver and sent upstream
1581 to the STREAM head.

1582 When a STREAMS-based device is opened, a STREAM shall be created that contains the
1583 STREAM head and the STREAM end (driver). If pipes are STREAMS-based in an
1584 implementation, when a pipe is created, two STREAMS shall be created, each containing a
1585 STREAM head. Other modules are added to the STREAM using *ioctl()*. New modules are
1586 “pushed” onto the STREAM one at a time in last-in, first-out (LIFO) style, as though the
1587 STREAM was a push-down stack.

1588 Priority

1589 Message types are classified according to their queuing priority and may be *normal* (non-
1590 priority), *priority*, or *high-priority* messages. A message belongs to a particular priority band that
1591 determines its ordering when placed on a queue. Normal messages have a priority band of 0 and
1592 shall always be placed at the end of the queue following all other messages in the queue. High-
1593 priority messages are always placed at the head of a queue, but shall be discarded if there is
1594 already a high-priority message in the queue. Their priority band shall be ignored; they are
1595 high-priority by virtue of their type. Priority messages have a priority band greater than 0.
1596 Priority messages are always placed after any messages of the same or higher priority. High-
1597 priority and priority messages are used to send control and data information outside the normal
1598 flow of control. By convention, high-priority messages shall not be affected by flow control.
1599 Normal and priority messages have separate flow controls.

1600 **Message Parts**

1601 A process may access STREAMS messages that contain a data part, control part, or both. The
 1602 data part is that information which is transmitted over the communication medium and the
 1603 control information is used by the local STREAMS modules. The other types of messages are
 1604 used between modules and are not accessible to processes. Messages containing only a data part
 1605 are accessible via *putmsg()*, *putpmsg()*, *getmsg()*, *getpmsg()*, *read()*, *readv()*, *write()*, or *writev()*.
 1606 Messages containing a control part with or without a data part are accessible via calls to
 1607 *putmsg()*, *putpmsg()*, *getmsg()*, or *getpmsg()*.

1608 **2.6.1 Accessing STREAMS**

1609 A process accesses STREAMS-based files using the standard functions *close()*, *ioctl()*, *getmsg()*,
 1610 *getpmsg()*, *open()*, *pipe()*, *poll()*, *putmsg()*, *putpmsg()*, *read()*, or *write()*. Refer to the applicable
 1611 function definitions for general properties and errors.

1612 Calls to *ioctl()* shall perform control functions on the STREAM associated with the file descriptor
 1613 *fildes*. The control functions may be performed by the STREAM head, a STREAMS module, or
 1614 the STREAMS driver for the STREAM.

1615 STREAMS modules and drivers can detect errors, sending an error message to the STREAM head,
 1616 thus causing subsequent functions to fail and set *errno* to the value specified in the
 1617 message. In addition, STREAMS modules and drivers can elect to fail a particular *ioctl()* request
 1618 alone by sending a negative acknowledgement message to the STREAM head. This shall cause
 1619 just the pending *ioctl()* request to fail and set *errno* to the value specified in the message.

1620 **2.7 XSI Interprocess Communication**

1621 XSI This section describes extensions to support interprocess communication. This functionality is
 1622 dependent on support of the XSI extension (and the rest of this section is not further shaded for
 1623 this option).

1624 The following message passing, semaphore, and shared memory services form an XSI
 1625 interprocess communication facility. Certain aspects of their operation are common, and are
 1626 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>	

1633 Another interprocess communication facility is provided by functions in the Realtime Option
 1634 Group; see Section 2.8 (on page 41).

1635 **2.7.1 IPC General Description**

1636 Each individual shared memory segment, message queue, and semaphore set shall be identified
 1637 by a unique positive integer, called, respectively, a shared memory identifier, *shmid*, a
 1638 semaphore identifier, *semid*, and a message queue identifier, *msqid*. The identifiers shall be
 1639 returned by calls to *shmget()*, *semget()*, and *msgget()*, respectively.

1640 Associated with each identifier is a data structure which contains data related to the operations
 1641 which may be or may have been performed; see the Base Definitions volume of
 1642 IEEE Std 1003.1-2001, <sys/shm.h>, <sys/sem.h>, and <sys/msg.h> for their descriptions.

1643 Each of the data structures contains both ownership information and an **ipc_perm** structure (see
 1644 the Base Definitions volume of IEEE Std 1003.1-2001, <sys/ipc.h>) which are used in conjunction
 1645 to determine whether or not read/write (read/alter for semaphores) permissions should be
 1646 granted to processes using the IPC facilities. The *mode* member of the **ipc_perm** structure acts as
 1647 a bit field which determines the permissions.

1648 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.

1657 The name of the **ipc_perm** structure is *shm_perm*, *sem_perm*, or *msg_perm*, depending on which
 1658 service is being used. In each case, read and write/alter permissions shall be granted to a process
 1659 if one or more of the following are true ("xxx" is replaced by *shm*, *sem*, or *msg*, as appropriate):

- 1660 • The process has appropriate privileges.
- 1661 • The effective user ID of the process matches *xxx_perm.cuid* or *xxx_perm.uid* in the data
 1662 structure associated with the IPC identifier, and the appropriate bit of the *user* field in
 1663 *xxx_perm.mode* is set.
- 1664 • The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* but the
 1665 effective group ID of the process matches *xxx_perm.cgid* or *xxx_perm.gid* in the data structure
 1666 associated with the IPC identifier, and the appropriate bit of the *group* field in *xxx_perm.mode*
 1667 is set.
- 1668 • The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* and the
 1669 effective group ID of the process does not match *xxx_perm.cgid* or *xxx_perm.gid* in the data
 1670 structure associated with the IPC identifier, but the appropriate bit of the *other* field in
 1671 *xxx_perm.mode* is set.

1672 Otherwise, the permission shall be denied.

1673 2.8 Realtime

1674 This section defines functions to support the source portability of applications with realtime
 1675 requirements. The presence of many of these functions is dependent on support for
 1676 implementation options described in the text.

1677 The specific functional areas included in this section and their scope include the following. Full
 1678 definitions of these terms can be found in the Base Definitions volume of IEEE Std 1003.1-2001,
 1679 Chapter 3, Definitions.

- 1680 • Semaphores
- 1681 • Process Memory Locking
- 1682 • Memory Mapped Files and Shared Memory Objects
- 1683 • Priority Scheduling
- 1684 • Realtime Signal Extension
- 1685 • Timers
- 1686 • Interprocess Communication
- 1687 • Synchronized Input and Output
- 1688 • Asynchronous Input and Output

1689 All the realtime functions defined in this volume of IEEE Std 1003.1-2001 are portable, although
 1690 some of the numeric parameters used by an implementation may have hardware dependencies.

1691 2.8.1 Realtime Signals

1692 RTS Realtime signal generation and delivery is dependent on support for the Realtime Signals
 1693 Extension option.

1694 See Section 2.4.2 (on page 29).

1695 2.8.2 Asynchronous I/O

1696 AIO The functionality described in this section is dependent on support of the Asynchronous Input
 1697 and Output option (and the rest of this section is not further shaded for this option).

1698 An asynchronous I/O control block structure **aiocb** is used in many asynchronous I/O
 1699 functions. It is defined in the Base Definitions volume of IEEE Std 1003.1-2001, **<aio.h>** and has
 1700 at least the following members:

1701	Member Type	Member Name	Description
1702	int	<i>aio_fildes</i>	File descriptor.
1703	off_t	<i>aio_offset</i>	File offset.
1704	volatile void*	<i>aio_buf</i>	Location of buffer.
1705	size_t	<i>aio_nbytes</i>	Length of transfer.
1706	int	<i>aio_reqprio</i>	Request priority offset.
1707	struct sigevent	<i>aio_sigevent</i>	Signal number and value.
1708	int	<i>aio_liq_opcode</i>	Operation to be performed.

1709 The *aio_fildes* element is the file descriptor on which the asynchronous operation is performed.

1710 If O_APPEND is not set for the file descriptor *aio_fildes* and if *aio_fildes* is associated with a
 1711 device that is capable of seeking, then the requested operation takes place at the absolute
 1712 position in the file as given by *aio_offset*, as if *Iseek()* were called immediately prior to the

1713 operation with an *offset* argument equal to *aio_offset* and a *whence* argument equal to SEEK_SET.
1714 If O_APPEND is set for the file descriptor, or if *aio_fildes* is associated with a device that is
1715 incapable of seeking, write operations append to the file in the same order as the calls were
1716 made, with the following exception: under implementation-defined circumstances, such as
1717 operation on a multi-processor or when requests of differing priorities are submitted at the same
1718 time, the ordering restriction may be relaxed. Since there is no way for a strictly conforming
1719 application to determine whether this relaxation applies, all strictly conforming applications
1720 which rely on ordering of output shall be written in such a way that they will operate correctly if
1721 the relaxation applies. After a successful call to enqueue an asynchronous I/O operation, the
1722 value of the file offset for the file is unspecified. The *aio_nbytes* and *aio_buf* elements are the same
1723 as the *nbyte* and *buf* arguments defined by *read()* and *write()*, respectively.

1724 If _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, then
1725 asynchronous I/O is queued in priority order, with the priority of each asynchronous operation
1726 based on the current scheduling priority of the calling process. The *aio_reqprio* member can be
1727 used to lower (but not raise) the asynchronous I/O operation priority and is within the range
1728 zero through {AIO_PRIO_DELTA_MAX}, inclusive. Unless both _POSIX_PRIORITIZED_IO and
1729 _POSIX_PRIORITY_SCHEDULING are defined, the order of processing asynchronous I/O
1730 requests is unspecified. When both _POSIX_PRIORITIZED_IO and
1731 _POSIX_PRIORITY_SCHEDULING are defined, the order of processing of requests submitted
1732 by processes whose schedulers are not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC is
1733 unspecified. The priority of an asynchronous request is computed as (process scheduling
1734 priority) minus *aio_reqprio*. The priority assigned to each asynchronous I/O request is an
1735 indication of the desired order of execution of the request relative to other asynchronous I/O
1736 requests for this file. If _POSIX_PRIORITIZED_IO is defined, requests issued with the same
1737 priority to a character special file are processed by the underlying device in FIFO order; the order
1738 of processing of requests of the same priority issued to files that are not character special files is
1739 unspecified. Numerically higher priority values indicate requests of higher priority. The value of
1740 *aio_reqprio* has no effect on process scheduling priority. When prioritized asynchronous I/O
1741 requests to the same file are blocked waiting for a resource required for that I/O operation, the
1742 higher-priority I/O requests shall be granted the resource before lower-priority I/O requests are
1743 granted the resource. The relative priority of asynchronous I/O and synchronous I/O is
1744 implementation-defined. If _POSIX_PRIORITIZED_IO is defined, the implementation shall
1745 define for which files I/O prioritization is supported.

1746 The *aio_sigevent* determines how the calling process shall be notified upon I/O completion, as
1747 specified in Section 2.4.1 (on page 28). If *aio_sigevent.sigev_notify* is SIGEV_NONE, then no signal
1748 shall be posted upon I/O completion, but the error status for the operation and the return status
1749 for the operation shall be set appropriately.

1750 The *aio_lio_opcode* field is used only by the *lio_listio()* call. The *lio_listio()* call allows multiple
1751 asynchronous I/O operations to be submitted at a single time. The function takes as an
1752 argument an array of pointers to **aiocb** structures. Each **aiocb** structure indicates the operation to
1753 be performed (read or write) via the *aio_lio_opcode* field.

1754 The address of the **aiocb** structure is used as a handle for retrieving the error status and return
1755 status of the asynchronous operation while it is in progress.

1756 The **aiocb** structure and the data buffers associated with the asynchronous I/O operation are
1757 being used by the system for asynchronous I/O while, and only while, the error status of the
1758 asynchronous operation is equal to [EINPROGRESS]. Applications shall not modify the **aiocb**
1759 structure while the structure is being used by the system for asynchronous I/O.

1760 The return status of the asynchronous operation is the number of bytes transferred by the I/O
1761 operation. If the error status is set to indicate an error completion, then the return status is set to

1762 the return value that the corresponding *read()*, *write()*, or *fsync()* call would have returned.
 1763 When the error status is not equal to [EINPROGRESS], the return status shall reflect the return
 1764 status of the corresponding synchronous operation.

1765 2.8.3 Memory Management

1766 2.8.3.1 Memory Locking

1767 MLR Range memory locking operations are defined in terms of pages. Implementations may restrict
 1768 the size and alignment of range lockings to be on page-size boundaries. The page size, in bytes,
 1769 is the value of the configurable system variable {PAGESIZE}. If an implementation has no
 1770 restrictions on size or alignment, it may specify a 1-byte page size.

1771 ML|MLR Memory locking guarantees the residence of portions of the address space. It is
 1772 implementation-defined whether locking memory guarantees fixed translation between virtual
 1773 addresses (as seen by the process) and physical addresses. Per-process memory locks are not
 1774 inherited across a *fork()*, and all memory locks owned by a process are unlocked upon *exec* or
 1775 process termination. Unmapping of an address range removes any memory locks established on
 1776 that address range by this process.

1777 2.8.3.2 Memory Mapped Files

1778 MF The functionality described in this section is dependent on support of the Memory Mapped Files
 1779 option (and the rest of this section is not further shaded for this option).

1780 Range memory mapping operations are defined in terms of pages. Implementations may
 1781 restrict the size and alignment of range mappings to be on page-size boundaries. The page size,
 1782 in bytes, is the value of the configurable system variable {PAGESIZE}. If an implementation has
 1783 no restrictions on size or alignment, it may specify a 1-byte page size.

1784 Memory mapped files provide a mechanism that allows a process to access files by directly
 1785 incorporating file data into its address space. Once a file is mapped into a process address space,
 1786 the data can be manipulated as memory. If more than one process maps a file, its contents are
 1787 shared among them. If the mappings allow shared write access, then data written into the
 1788 memory object through the address space of one process appears in the address spaces of all
 1789 processes that similarly map the same portion of the memory object.

1790 SHM Shared memory objects are named regions of storage that may be independent of the file system
 1791 and can be mapped into the address space of one or more processes to allow them to share the
 1792 associated memory.

1793 SHM An *unlink()* of a file or *shm_unlink()* of a shared memory object, while causing the removal of the
 1794 name, does not unmap any mappings established for the object. Once the name has been
 1795 removed, the contents of the memory object are preserved as long as it is referenced. The
 1796 memory object remains referenced as long as a process has the memory object open or has some
 1797 area of the memory object mapped.

1798 2.8.3.3 Memory Protection

1799 MPR MF The functionality described in this section is dependent on support of the Memory Protection
 1800 and Memory Mapped Files option (and the rest of this section is not further shaded for these
 1801 options).

1802 When an object is mapped, various application accesses to the mapped region may result in
 1803 signals. In this context, SIGBUS is used to indicate an error using the mapped object, and
 1804 SIGSEGV is used to indicate a protection violation or misuse of an address:

- 1805 • A mapping may be restricted to disallow some types of access.
1806 • Write attempts to memory that was mapped without write access, or any access to memory
1807 mapped PROT_NONE, shall result in a SIGSEGV signal.
1808 • References to unmapped addresses shall result in a SIGSEGV signal.
1809 • Reference to whole pages within the mapping, but beyond the current length of the object,
1810 shall result in a SIGBUS signal.
1811 • The size of the object is unaffected by access beyond the end of the object (even if a SIGBUS is
1812 not generated).

1813 **2.8.3.4 Typed Memory Objects**

1814 **TYM** The functionality described in this section is dependent on support of the Typed Memory
1815 Objects option (and the rest of this section is not further shaded for this option).

1816 Implementations may support the Typed Memory Objects option without supporting the
1817 Memory Mapped Files option or the Shared Memory Objects option. Typed memory objects are
1818 implementation-configurable named storage pools accessible from one or more processors in a
1819 system, each via one or more ports, such as backplane buses, LANs, I/O channels, and so on.
1820 Each valid combination of a storage pool and a port is identified through a name that is defined
1821 at system configuration time, in an implementation-defined manner; the name may be
1822 independent of the file system. Using this name, a typed memory object can be opened and
1823 mapped into process address space. For a given storage pool and port, it is necessary to support
1824 both dynamic allocation from the pool as well as mapping at an application-supplied offset
1825 within the pool; when dynamic allocation has been performed, subsequent deallocation must be
1826 supported. Lastly, accessing typed memory objects from different ports requires a method for
1827 obtaining the offset and length of contiguous storage of a region of typed memory (dynamically
1828 allocated or not); this allows typed memory to be shared among processes and/or processors
1829 while being accessed from the desired port.

1830 **2.8.4 Process Scheduling**

1831 **PS** The functionality described in this section is dependent on support of the Process Scheduling
1832 option (and the rest of this section is not further shaded for this option).

1833 **Scheduling Policies**

1834 The scheduling semantics described in this volume of IEEE Std 1003.1-2001 are defined in terms
1835 of a conceptual model that contains a set of thread lists. No implementation structures are
1836 necessarily implied by the use of this conceptual model. It is assumed that no time elapses
1837 during operations described using this model, and therefore no simultaneous operations are
1838 possible. This model discusses only processor scheduling for runnable threads, but it should be
1839 noted that greatly enhanced predictability of realtime applications results if the sequencing of
1840 other resources takes processor scheduling policy into account.

1841 There is, conceptually, one thread list for each priority. A runnable thread will be on the thread
1842 list for that thread's priority. Multiple scheduling policies shall be provided. Each non-empty
1843 thread list is ordered, contains a head as one end of its order, and a tail as the other. The purpose
1844 of a scheduling policy is to define the allowable operations on this set of lists (for example,
1845 moving threads between and within lists).

1846 Each process shall be controlled by an associated scheduling policy and priority. These
1847 parameters may be specified by explicit application execution of the *sched_setscheduler()* or
1848 *sched_setparam()* functions.

1849 Each thread shall be controlled by an associated scheduling policy and priority. These
1850 parameters may be specified by explicit application execution of the *pthread_setschedparam()*
1851 function.

1852 Associated with each policy is a priority range. Each policy definition shall specify the minimum
1853 priority range for that policy. The priority ranges for each policy may but need not overlap the
1854 priority ranges of other policies.

1855 A conforming implementation shall select the thread that is defined as being at the head of the
1856 highest priority non-empty thread list to become a running thread, regardless of its associated
1857 policy. This thread is then removed from its thread list.

1858 Four scheduling policies are specifically required. Other implementation-defined scheduling
1859 policies may be defined. The following symbols are defined in the Base Definitions volume of
1860 IEEE Std 1003.1-2001, <sched.h>:

1861 SCHED_FIFO First in, first out (FIFO) scheduling policy.

1862 SCHED_RR Round robin scheduling policy.

1863 SS SCHED_SPORADIC Sporadic server scheduling policy.

1864 SCHED_OTHER Another scheduling policy.

1865 The values of these symbols shall be distinct.

1866 **SCHED_FIFO**

1867 Conforming implementations shall include a scheduling policy called the FIFO scheduling
1868 policy.

1869 Threads scheduled under this policy are chosen from a thread list that is ordered by the time its
1870 threads have been on the list without being executed; generally, the head of the list is the thread
1871 that has been on the list the longest time, and the tail is the thread that has been on the list the
1872 shortest time.

1873 Under the SCHED_FIFO policy, the modification of the definitional thread lists is as follows:

- 1874 1. When a running thread becomes a preempted thread, it becomes the head of the thread list
1875 for its priority.
- 1876 2. When a blocked thread becomes a runnable thread, it becomes the tail of the thread list for
1877 its priority.
- 1878 3. When a running thread calls the *sched_setscheduler()* function, the process specified in the
1879 function call is modified to the specified policy and the priority specified by the *param*
1880 argument.
- 1881 4. When a running thread calls the *sched_setparam()* function, the priority of the process
1882 specified in the function call is modified to the priority specified by the *param* argument.
- 1883 5. When a running thread calls the *pthread_setschedparam()* function, the thread specified in
1884 the function call is modified to the specified policy and the priority specified by the *param*
1885 argument.
- 1886 6. When a running thread calls the *pthread_setschedprio()* function, the thread specified in the
1887 function call is modified to the priority specified by the *prio* argument.
- 1888 7. If a thread whose policy or priority has been modified other than by *pthread_setschedprio()*
1889 is a running thread or is runnable, it then becomes the tail of the thread list for its new
1890 priority.

- 1891 8. If a thread whose policy or priority has been modified by *pthread_setschedprio()* is a
1892 running thread or is runnable, the effect on its position in the thread list depends on the
1893 direction of the modification, as follows:
1894 a. If the priority is raised, the thread becomes the tail of the thread list.
1895 b. If the priority is unchanged, the thread does not change position in the thread list.
1896 c. If the priority is lowered, the thread becomes the head of the thread list.
1897 9. When a running thread issues the *sched_yield()* function, the thread becomes the tail of the
1898 thread list for its priority.
1899 10. At no other time is the position of a thread with this scheduling policy within the thread
1900 lists affected.

1901 For this policy, valid priorities shall be within the range returned by the *sched_get_priority_max()*
1902 and *sched_get_priority_min()* functions when *SCHED_FIFO* is provided as the parameter.
1903 Conforming implementations shall provide a priority range of at least 32 priorities for this
1904 policy.

1905 **SCHED_RR**

1906 Conforming implementations shall include a scheduling policy called the “round robin”
1907 scheduling policy. This policy shall be identical to the *SCHED_FIFO* policy with the additional
1908 condition that when the implementation detects that a running thread has been executing as a
1909 running thread for a time period of the length returned by the *sched_rr_get_interval()* function or
1910 longer, the thread shall become the tail of its thread list and the head of that thread list shall be
1911 removed and made a running thread.

1912 The effect of this policy is to ensure that if there are multiple *SCHED_RR* threads at the same
1913 priority, one of them does not monopolize the processor. An application should not rely only on
1914 the use of *SCHED_RR* to ensure application progress among multiple threads if the application
1915 includes threads using the *SCHED_FIFO* policy at the same or higher priority levels or
1916 *SCHED_RR* threads at a higher priority level.

1917 A thread under this policy that is preempted and subsequently resumes execution as a running
1918 thread completes the unexpired portion of its round robin interval time period.

1919 For this policy, valid priorities shall be within the range returned by the *sched_get_priority_max()*
1920 and *sched_get_priority_min()* functions when *SCHED_RR* is provided as the parameter.
1921 Conforming implementations shall provide a priority range of at least 32 priorities for this
1922 policy.

1923 **SCHED_SPORADIC**

1924 SS|TSP The functionality described in this section is dependent on support of the Process Sporadic
1925 Server or Thread Sporadic Server options (and the rest of this section is not further shaded for
1926 these options).

1927 If *_POSIX_SPORADIC_SERVER* or *_POSIX_THREAD_SPORADIC_SERVER* is defined, the
1928 implementation shall include a scheduling policy identified by the value *SCHED_SPORADIC*.

1929 The sporadic server policy is based primarily on two parameters: the replenishment period and
1930 the available execution capacity. The replenishment period is given by the *sched_ss_repl_period*
1931 member of the *sched_param* structure. The available execution capacity is initialized to the
1932 value given by the *sched_ss_init_budget* member of the same parameter. The sporadic server
1933 policy is identical to the *SCHED_FIFO* policy with some additional conditions that cause the
1934 thread’s assigned priority to be switched between the values specified by the *sched_priority* and

1935 *sched_ss_low_priority* members of the **sched_param** structure.

1936 The priority assigned to a thread using the sporadic server scheduling policy is determined in
 1937 the following manner: if the available execution capacity is greater than zero and the number of
 1938 pending replenishment operations is strictly less than *sched_ss_max_repl*, the thread is assigned
 1939 the priority specified by *sched_priority*; otherwise, the assigned priority shall be
 1940 *sched_ss_low_priority*. If the value of *sched_priority* is less than or equal to the value of
 1941 *sched_ss_low_priority*, the results are undefined. When active, the thread shall belong to the
 1942 thread list corresponding to its assigned priority level, according to the mentioned priority
 1943 assignment. The modification of the available execution capacity and, consequently of the
 1944 assigned priority, is done as follows:

- 1945 1. When the thread at the head of the *sched_priority* list becomes a running thread, its
 1946 execution time shall be limited to at most its available execution capacity, plus the
 1947 resolution of the execution time clock used for this scheduling policy. This resolution shall
 1948 be implementation-defined.
- 1949 2. Each time the thread is inserted at the tail of the list associated with *sched_priority*—
 1950 because as a blocked thread it became runnable with priority *sched_priority* or because a
 1951 replenishment operation was performed—the time at which this operation is done is
 1952 posted as the *activation_time*.
- 1953 3. When the running thread with assigned priority equal to *sched_priority* becomes a
 1954 preempted thread, it becomes the head of the thread list for its priority, and the execution
 1955 time consumed is subtracted from the available execution capacity. If the available
 1956 execution capacity would become negative by this operation, it shall be set to zero.
- 1957 4. When the running thread with assigned priority equal to *sched_priority* becomes a blocked
 1958 thread, the execution time consumed is subtracted from the available execution capacity,
 1959 and a replenishment operation is scheduled, as described in 6 and 7. If the available
 1960 execution capacity would become negative by this operation, it shall be set to zero.
- 1961 5. When the running thread with assigned priority equal to *sched_priority* reaches the limit
 1962 imposed on its execution time, it becomes the tail of the thread list for
 1963 *sched_ss_low_priority*, the execution time consumed is subtracted from the available
 1964 execution capacity (which becomes zero), and a replenishment operation is scheduled, as
 1965 described in 6 and 7.
- 1966 6. Each time a replenishment operation is scheduled, the amount of execution capacity to be
 1967 replenished, *replenish_amount*, is set equal to the execution time consumed by the thread
 1968 since the *activation_time*. The replenishment is scheduled to occur at *activation_time* plus
 1969 *sched_ss_repl_period*. If the scheduled time obtained is before the current time, the
 1970 replenishment operation is carried out immediately. Several replenishment operations may
 1971 be pending at the same time, each of which will be serviced at its respective scheduled
 1972 time. With the above rules, the number of replenishment operations simultaneously
 1973 pending for a given thread that is scheduled under the sporadic server policy shall not be
 1974 greater than *sched_ss_max_repl*.
- 1975 7. A replenishment operation consists of adding the corresponding *replenish_amount* to the
 1976 available execution capacity at the scheduled time. If, as a consequence of this operation,
 1977 the execution capacity would become larger than *sched_ss_initial_budget*, it shall be
 1978 rounded down to a value equal to *sched_ss_initial_budget*. Additionally, if the thread was
 1979 runnable or running, and had assigned priority equal to *sched_ss_low_priority*, then it
 1980 becomes the tail of the thread list for *sched_priority*.

1981 Execution time is defined in Section 2.2.2 (on page 14).

1982 For this policy, changing the value of a CPU-time clock via *clock_settime()* shall have no effect on
1983 its behavior.

1984 For this policy, valid priorities shall be within the range returned by the *sched_get_priority_min()*
1985 and *sched_get_priority_max()* functions when SCHED_SPORADIC is provided as the parameter.
1986 Conforming implementations shall provide a priority range of at least 32 distinct priorities for
1987 this policy.

1988 SCHED_OTHER

1989 Conforming implementations shall include one scheduling policy identified as SCHED_OTHER
1990 (which may execute identically with either the FIFO or round robin scheduling policy). The
1991 effect of scheduling threads with the SCHED_OTHER policy in a system in which other threads
1992 are executing under SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC is implementation-
1993 defined.

1994 This policy is defined to allow strictly conforming applications to be able to indicate in a
1995 portable manner that they no longer need a realtime scheduling policy.

1996 For threads executing under this policy, the implementation shall use only priorities within the
1997 range returned by the *sched_get_priority_max()* and *sched_get_priority_min()* functions when
1998 SCHED_OTHER is provided as the parameter.

1999 2.8.5 Clocks and Timers

2000 TMR The functionality described in this section is dependent on support of the Timers option (and the
2001 rest of this section is not further shaded for this option).

2002 The <time.h> header defines the types and manifest constants used by the timing facility.

2003 Time Value Specification Structures

2004 Many of the timing facility functions accept or return time value specifications. A time value
2005 structure **timespec** specifies a single time value and includes at least the following members:

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

2010 The *tv_nsec* member is only valid if greater than or equal to zero, and less than the number of
2011 nanoseconds in a second (1 000 million). The time interval described by this structure is (*tv_sec* *
2012 $10^9 + tv_nsec$) nanoseconds.

2013 A time value structure **itimerspec** specifies an initial timer value and a repetition interval for use
2014 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.

2019 If the value described by *it_value* is non-zero, it indicates the time to or time of the next timer
2020 expiration (for relative and absolute timer values, respectively). If the value described by *it_value*
2021 is zero, the timer shall be disarmed.

2022 If the value described by *it_interval* is non-zero, it specifies an interval which shall be used in
2023 reloading the timer when it expires; that is, a periodic timer is specified. If the value described by

2024 *it_interval* is zero, the timer is disarmed after its next expiration; that is, a one-shot timer is
 2025 specified.

2026 Timer Event Notification Control Block

2027 RTS Per-process timers may be created that notify the process of timer expirations by queuing a
 2028 realtime extended signal. The **sigevent** structure, defined in the Base Definitions volume of
 2029 IEEE Std 1003.1-2001, **<signal.h>**, is used in creating such a timer. The **sigevent** structure
 2030 contains the signal number and an application-specific data value which shall be used when
 2031 notifying the calling process of timer expiration events.

2032 Manifest Constants

2033 The following constants are defined in the Base Definitions volume of IEEE Std 1003.1-2001,
 2034 **<time.h>**:

2035	CLOCK_REALTIME	The identifier for the system-wide realtime clock.
2036	TIMER_ABSTIME	Flag indicating time is absolute with respect to the clock associated with a timer.
2038 MON	CLOCK_MONOTONIC	The identifier for the system-wide monotonic clock, which is defined as a clock whose value cannot be set via <i>clock_settime()</i> and which cannot have backward clock jumps. The maximum possible clock jump is implementation-defined.

2042 MON The maximum allowable resolution for CLOCK_REALTIME and CLOCK_MONOTONIC clocks
 2043 and all time services based on these clocks is represented by **{POSIX_CLOCKRES_MIN}** and
 2044 shall be defined as 20 ms (1/50 of a second). Implementations may support smaller values of
 2045 resolution for these clocks to provide finer granularity time bases. The actual resolution
 2046 supported by an implementation for a specific clock is obtained using the *clock_getres()* function.
 2047 If the actual resolution supported for a time service based on one of these clocks differs from the
 2048 resolution supported for that clock, the implementation shall document this difference.

2049 MON The minimum allowable maximum value for CLOCK_REALTIME and CLOCK_MONOTONIC
 2050 clocks and all absolute time services based on them is the same as that defined by the ISO C
 2051 standard for the **time_t** type. If the maximum value supported by a time service based on one of
 2052 these clocks differs from the maximum value supported by that clock, the implementation shall
 2053 document this difference.

2054 Execution Time Monitoring

2055 CPT If **_POSIX_CPUTIME** is defined, process CPU-time clocks shall be supported in addition to the
 2056 clocks described in **Manifest Constants**.

2057 TCT If **_POSIX_THREAD_CPUTIME** is defined, thread CPU-time clocks shall be supported.

2058 CPT|TCT CPU-time clocks measure execution or CPU time, which is defined in the Base Definitions
 2059 volume of IEEE Std 1003.1-2001, Section 3.117, CPU Time (Execution Time). The mechanism
 2060 used to measure execution time is described in the Base Definitions volume of
 2061 IEEE Std 1003.1-2001, Section 4.9, Measurement of Execution Time.

2062 CPT If **_POSIX_CPUTIME** is defined, the following constant of the type **clockid_t** is defined in
 2063 **<time.h>**:

2064 **CLOCK_PROCESS_CPUTIME_ID**

2065 When this value of the type **clockid_t** is used in a *clock()* or *timer*()* function call, it is
 2066 interpreted as the identifier of the CPU-time clock associated with the process making the

2067 function call.

2068

2069 TCT If `_POSIX_THREAD_CPUTIME` is defined, the following constant of the type `clockid_t` is
2070 defined in `<time.h>`:

2071 **CLOCK_THREAD_CPUTIME_ID**

2072 When this value of the type `clockid_t` is used in a `clock()` or `timer*`() function call, it is
2073 interpreted as the identifier of the CPU-time clock associated with the thread making the
2074 function call.

2075 2.9 Threads

2076 THR The functionality described in this section is dependent on support of the Threads option (and
2077 the rest of this section is not further shaded for this option).

2078 This section defines functionality to support multiple flows of control, called “threads”, within a
2079 process. For the definition of threads, see the Base Definitions volume of IEEE Std 1003.1-2001,
2080 Section 3.393, Thread.

2081 The specific functional areas covered by threads and their scope include:

- 2082 • Thread management: the creation, control, and termination of multiple flows of control in the
2083 same process under the assumption of a common shared address space
- 2084 • Synchronization primitives optimized for tightly coupled operation of multiple control flows
2085 in a common, shared address space

2086 2.9.1 Thread-Safety

2087 All functions defined by this volume of IEEE Std 1003.1-2001 shall be thread-safe, except that the
2088 following functions¹ need not be thread-safe.

2089 <code>asctime()</code>	2089 <code>ecvt()</code>	2089 <code>gethostent()</code>	2089 <code>getutxline()</code>	2089 <code>putc_unlocked()</code>
2090 <code>basename()</code>	2090 <code>encrypt()</code>	2090 <code>getlogin()</code>	2090 <code>gmtime()</code>	2090 <code>putchar_unlocked()</code>
2091 <code>catgets()</code>	2091 <code>endgrent()</code>	2091 <code>getnetbyaddr()</code>	2091 <code>hcreate()</code>	2091 <code>putenv()</code>
2092 <code>crypt()</code>	2092 <code>endpwent()</code>	2092 <code>getnetbyname()</code>	2092 <code>hdestroy()</code>	2092 <code>pututxline()</code>
2093 <code>ctime()</code>	2093 <code>endutxent()</code>	2093 <code>getnetent()</code>	2093 <code>hsearch()</code>	2093 <code>rand()</code>
2094 <code>dbm_clearerr()</code>	2094 <code>fcvt()</code>	2094 <code> getopt()</code>	2094 <code>inet_ntoa()</code>	2094 <code>readdir()</code>
2095 <code>dbm_close()</code>	2095 <code>ftw()</code>	2095 <code>getprotobyname()</code>	2095 <code>l64a()</code>	2095 <code>setenv()</code>
2096 <code>dbm_delete()</code>	2096 <code>gcvt()</code>	2096 <code>getprotobynumber()</code>	2096 <code>lgamma()</code>	2096 <code>setgrent()</code>
2097 <code>dbm_error()</code>	2097 <code>getc_unlocked()</code>	2097 <code>getprotoent()</code>	2097 <code>lgammaf()</code>	2097 <code>setkey()</code>
2098 <code>dbm_fetch()</code>	2098 <code>getchar_unlocked()</code>	2098 <code>getpwent()</code>	2098 <code>lgammal()</code>	2098 <code>setpwent()</code>
2099 <code>dbm_firstkey()</code>	2099 <code>getdate()</code>	2099 <code>getpwnam()</code>	2099 <code>localeconv()</code>	2099 <code>setutxent()</code>
2100 <code>dbm_nextkey()</code>	2100 <code>getenv()</code>	2100 <code>getpwuid()</code>	2100 <code>localtime()</code>	2100 <code>strerror()</code>
2101 <code>dbm_open()</code>	2101 <code>getgrent()</code>	2101 <code>getservbyname()</code>	2101 <code>lrand48()</code>	2101 <code>strtok()</code>
2102 <code>dbm_store()</code>	2102 <code>getgrgid()</code>	2102 <code>getservbyport()</code>	2102 <code>mrand48()</code>	2102 <code>ttyname()</code>
2103 <code>dirname()</code>	2103 <code>getgrnam()</code>	2103 <code>getservent()</code>	2103 <code>nftw()</code>	2103 <code>unsetenv()</code>

2104 _____

2105 1. The functions in the table are not shaded to denote applicable options. Individual reference pages should be consulted.

2106	<i>dllerror()</i>	<i>gethostbyaddr()</i>	<i>getutxent()</i>	<i>nl_langinfo()</i>	<i>wcstombs()</i>
2107	<i>drand48()</i>	<i>gethostbyname()</i>	<i>getutxid()</i>	<i>ptsname()</i>	<i>wctomb()</i>

2108 The *ctermid()* and *tmpnam()* functions need not be thread-safe if passed a NULL argument. The
 2109 *wcrtomb()* and *wcsrtombs()* functions need not be thread-safe if passed a NULL *ps* argument.

2110 Implementations shall provide internal synchronization as necessary in order to satisfy this
 2111 requirement.

2112 2.9.2 Thread IDs

2113 Although implementations may have thread IDs that are unique in a system, applications
 2114 should only assume that thread IDs are usable and unique within a single process. The effect of
 2115 calling any of the functions defined in this volume of IEEE Std 1003.1-2001 and passing as an
 2116 argument the thread ID of a thread from another process is unspecified. A conforming
 2117 implementation is free to reuse a thread ID after the thread terminates if it was created with the
 2118 *detachstate* attribute set to PTHREAD_CREATE_DETACHED or if *pthread_detach()* or
 2119 *pthread_join()* has been called for that thread. If a thread is detached, its thread ID is invalid for
 2120 use as an argument in a call to *pthread_detach()* or *pthread_join()*.

2121 2.9.3 Thread Mutexes

2122 A thread that has blocked shall not prevent any unblocked thread that is eligible to use the same
 2123 processing resources from eventually making forward progress in its execution. Eligibility for
 2124 processing resources is determined by the scheduling policy.

2125 A thread shall become the owner of a mutex, *m*, when one of the following occurs:

- 2126 • It returns successfully from *pthread_mutex_lock()* with *m* as the *mutex* argument.
- 2127 • It returns successfully from *pthread_mutex_trylock()* with *m* as the *mutex* argument.
- 2128 TMO • It returns successfully from *pthread_mutex_timedlock()* with *m* as the *mutex* argument.
- 2129 • It returns (successfully or not) from *pthread_cond_wait()* with *m* as the *mutex* argument
 (except as explicitly indicated otherwise for certain errors).
- 2132 • It returns (successfully or not) from *pthread_cond_timedwait()* with *m* as the *mutex* argument
 (except as explicitly indicated otherwise for certain errors).

2133 The thread shall remain the owner of *m* until one of the following occurs:

- 2134 • It executes *pthread_mutex_unlock()* with *m* as the *mutex* argument
- 2135 • It blocks in a call to *pthread_cond_wait()* with *m* as the *mutex* argument.
- 2136 • It blocks in a call to *pthread_cond_timedwait()* with *m* as the *mutex* argument.

2137 The implementation shall behave as if at all times there is at most one owner of any mutex.

2138 A thread that becomes the owner of a mutex is said to have “acquired” the mutex and the mutex
 2139 is said to have become “locked”; when a thread gives up ownership of a mutex it is said to have
 2140 “released” the mutex and the mutex is said to have become “unlocked”.

2.9.4 Thread Scheduling

2141 TPS The functionality described in this section is dependent on support of the Thread Execution 1
2143 Scheduling option (and the rest of this section is not further shaded for this option).

2144 Thread Scheduling Attributes

2145 In support of the scheduling function, threads have attributes which are accessed through the 1
2146 `pthread_attr_t` thread creation attributes object.

2147 The `contentionscope` attribute defines the scheduling contention scope of the thread to be either 1
2148 PTHREAD_SCOPE_PROCESS or PTHREAD_SCOPE_SYSTEM.

2149 The `inheritsched` attribute specifies whether a newly created thread is to inherit the scheduling 1
2150 attributes of the creating thread or to have its scheduling values set according to the other
2151 scheduling attributes in the `pthread_attr_t` object.

2152 The `schedpolicy` attribute defines the scheduling policy for the thread. The `schedparam` attribute 1
2153 defines the scheduling parameters for the thread. The interaction of threads having different
2154 policies within a process is described as part of the definition of those policies.

2155 If the Thread Execution Scheduling option is defined, and the `schedpolicy` attribute specifies one 1
2156 of the priority-based policies defined under this option, the `schedparam` attribute contains the
2157 scheduling priority of the thread. A conforming implementation ensures that the priority value
2158 in `schedparam` is in the range associated with the scheduling policy when the thread attributes
2159 object is used to create a thread, or when the scheduling attributes of a thread are dynamically
2160 modified. The meaning of the priority value in `schedparam` is the same as that of `priority`.

2161 TSP If `_POSIX_THREAD_SPORADIC_SERVER` is defined, the `schedparam` attribute supports four 1
2162 new members that are used for the sporadic server scheduling policy. These members are
2163 `sched_ss_low_priority`, `sched_ss_repl_period`, `sched_ss_init_budget`, and `sched_ss_max_repl`. The
2164 meaning of these attributes is the same as in the definitions that appear under Section 2.8.4 (on
2165 page 44).

2166 When a process is created, its single thread has a scheduling policy and associated attributes 1
2167 equal to the process' policy and attributes. The default scheduling contention scope value is
2168 implementation-defined. The default values of other scheduling attributes are implementation-
2169 defined.

2170 Thread Scheduling Contention Scope

2171 The scheduling contention scope of a thread defines the set of threads with which the thread 1
2172 competes for use of the processing resources. The scheduling operation selects at most one
2173 thread to execute on each processor at any point in time and the thread's scheduling attributes
2174 (for example, `priority`), whether under process scheduling contention scope or system scheduling
2175 contention scope, are the parameters used to determine the scheduling decision.

2176 The scheduling contention scope, in the context of scheduling a mixed scope environment,
2177 affects threads as follows:

- 2178 • A thread created with PTHREAD_SCOPE_SYSTEM scheduling contention scope contends
2179 for resources with all other threads in the same scheduling allocation domain relative to their
2180 system scheduling attributes. The system scheduling attributes of a thread created with
2181 PTHREAD_SCOPE_SYSTEM scheduling contention scope are the scheduling attributes with
2182 which the thread was created. The system scheduling attributes of a thread created with
2183 PTHREAD_SCOPE_PROCESS scheduling contention scope are the implementation-defined
2184 mapping into system attribute space of the scheduling attributes with which the thread was
2185 created.

- 2186 • Threads created with PTHREAD_SCOPE_PROCESS scheduling contention scope contend
2187 directly with other threads within their process that were created with
2188 PTHREAD_SCOPE_PROCESS scheduling contention scope. The contention is resolved
2189 based on the threads' scheduling attributes and policies. It is unspecified how such threads
2190 are scheduled relative to threads in other processes or threads with
2191 PTHREAD_SCOPE_SYSTEM scheduling contention scope.
- 2192 • Conforming implementations shall support the PTHREAD_SCOPE_PROCESS scheduling
2193 contention scope, the PTHREAD_SCOPE_SYSTEM scheduling contention scope, or both.

2194 **Scheduling Allocation Domain**

2195 Implementations shall support scheduling allocation domains containing one or more
2196 processors. It should be noted that the presence of multiple processors does not automatically
2197 indicate a scheduling allocation domain size greater than one. Conforming implementations on
2198 multi-processors may map all or any subset of the CPUs to one or multiple scheduling allocation
2199 domains, and could define these scheduling allocation domains on a per-thread, per-process, or
2200 per-system basis, depending on the types of applications intended to be supported by the
2201 implementation. The scheduling allocation domain is independent of scheduling contention
2202 scope, as the scheduling contention scope merely defines the set of threads with which a thread
2203 contends for processor resources, while scheduling allocation domain defines the set of
2204 processors for which it contends. The semantics of how this contention is resolved among
2205 threads for processors is determined by the scheduling policies of the threads.

2206 The choice of scheduling allocation domain size and the level of application control over
2207 scheduling allocation domains is implementation-defined. Conforming implementations may
2208 change the size of scheduling allocation domains and the binding of threads to scheduling
2209 allocation domains at any time.

2210 For application threads with scheduling allocation domains of size equal to one, the scheduling
2211 rules defined for SCHED_FIFO and SCHED_RR shall be used; see **Scheduling Policies** (on page
2212 44). All threads with system scheduling contention scope, regardless of the processes in which
2213 they reside, compete for the processor according to their priorities. Threads with process
2214 scheduling contention scope compete only with other threads with process scheduling
2215 contention scope within their process.

2216 For application threads with scheduling allocation domains of size greater than one, the rules
2217 TSP defined for SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC shall be used in an
2218 implementation-defined manner. Each thread with system scheduling contention scope
2219 competes for the processors in its scheduling allocation domain in an implementation-defined
2220 manner according to its priority. Threads with process scheduling contention scope are
2221 scheduled relative to other threads within the same scheduling contention scope in the process.

2222 TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, the rules defined for SCHED_SPORADIC
2223 in **Scheduling Policies** (on page 44) shall be used in an implementation-defined manner for
2224 application threads whose scheduling allocation domain size is greater than one.

2225 **Scheduling Documentation**

2226 TSP If `_POSIX_PRIORITY_SCHEDULING` is defined, then any scheduling policies beyond
2227 SCED_OTHER, SCED_FIFO, SCED_RR, and SCED_SPORADIC, as well as the effects of
2228 the scheduling policies indicated by these other values, and the attributes required in order to
2229 support such a policy, are implementation-defined. Furthermore, the implementation shall
2230 document the effect of all processor scheduling allocation domain values supported for these
2231 policies.

2232 **2.9.5 Thread Cancellation**

2233 The thread cancellation mechanism allows a thread to terminate the execution of any other
2234 thread in the process in a controlled manner. The target thread (that is, the one that is being
2235 canceled) is allowed to hold cancellation requests pending in a number of ways and to perform
2236 application-specific cleanup processing when the notice of cancellation is acted upon.

2237 Cancellation is controlled by the cancellation control functions. Each thread maintains its own
2238 cancelability state. Cancellation may only occur at cancellation points or when the thread is
2239 asynchronously cancelable.

2240 The thread cancellation mechanism described in this section depends upon programs having set
2241 *deferred* cancelability state, which is specified as the default. Applications shall also carefully
2242 follow static lexical scoping rules in their execution behavior. For example, use of `setjmp()`,
2243 `return`, `goto`, and so on, to leave user-defined cancellation scopes without doing the necessary
2244 scope pop operation results in undefined behavior.

2245 Use of asynchronous cancelability while holding resources which potentially need to be released
2246 may result in resource loss. Similarly, cancellation scopes may only be safely manipulated
2247 (pushed and popped) when the thread is in the *deferred* or *disabled* cancelability states.

2248 **2.9.5.1 Cancelability States**

2249 The cancelability state of a thread determines the action taken upon receipt of a cancellation
2250 request. The thread may control cancellation in a number of ways.

2251 Each thread maintains its own cancelability state, which may be encoded in two bits:

- 2252 1. Cancelability-Enable: When cancelability is `PTHREAD_CANCEL_DISABLE` (as defined in
2253 the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`), cancellation requests
2254 against the target thread are held pending. By default, cancelability is set to
2255 `PTHREAD_CANCEL_ENABLE` (as defined in `<pthread.h>`).
- 2256 2. Cancelability Type: When cancelability is enabled and the cancelability type is
2257 `PTHREAD_CANCEL_ASYNCHRONOUS` (as defined in `<pthread.h>`), new or pending
2258 cancellation requests may be acted upon at any time. When cancelability is enabled and
2259 the cancelability type is `PTHREAD_CANCEL_DEFERRED` (as defined in `<pthread.h>`),
2260 cancellation requests are held pending until a cancellation point (see below) is reached. If
2261 cancelability is disabled, the setting of the cancelability type has no immediate effect as all
2262 cancellation requests are held pending; however, once cancelability is enabled again the
2263 new type is in effect. The cancelability type is `PTHREAD_CANCEL_DEFERRED` in all
2264 newly created threads including the thread in which `main()` was first invoked.

2265 2.9.5.2 Cancellation Points

2266 Cancellation points shall occur when a thread is executing the following functions:

2267	<i>accept()</i>	<i>mq_timedsend()</i>	<i>putpmsg()</i>	<i>sigtimedwait()</i>	
2268	<i>aio_suspend()</i>	<i>msgrcv()</i>	<i>pwrite()</i>	<i>sigwait()</i>	
2269	<i>clock_nanosleep()</i>	<i>msgsnd()</i>	<i>read()</i>	<i>sigwaitinfo()</i>	
2270	<i>close()</i>	<i>msync()</i>	<i>readv()</i>	<i>sleep()</i>	
2271	<i>connect()</i>	<i>nanosleep()</i>	<i>recv()</i>	<i>system()</i>	
2272	<i>creat()</i>	<i>open()</i>	<i>recvfrom()</i>	<i>tcdrain()</i>	
2273	<i>fcntl()</i> †	<i>pause()</i>	<i>recvmsg()</i>	<i>usleep()</i>	2
2274	<i>fdatasync()</i>	<i>poll()</i>	<i>select()</i>	<i>wait()</i>	
2275	<i>fsync()</i>	<i>pread()</i>	<i>sem_timedwait()</i>	<i>waitid()</i>	
2276	<i>getmsg()</i>	<i>pselect()</i>	<i>sem_wait()</i>	<i>waitpid()</i>	1
2277	<i>getpmsg()</i>	<i>pthread_cond_timedwait()</i>	<i>send()</i>	<i>write()</i>	
2278	<i>lockf()</i>	<i>pthread_cond_wait()</i>	<i>sendmsg()</i>	<i>writev()</i>	
2279	<i>mq_receive()</i>	<i>pthread_join()</i>	<i>sendto()</i>		
2280	<i>mq_send()</i>	<i>pthread_testcancel()</i>	<i>sigpause()</i>		
2281	<i>mq_timedreceive()</i>	<i>putmsg()</i>	<i>sigsuspend()</i>		

2282 _____

2283 † When the *cmd* argument is F_SETLKW.

2284 A cancellation point may also occur when a thread is executing the following functions:

2285	access()	fstat()	getwd()	putc_unlocked()	2
2286	asctime()	ftell()	glob()	putchar()	2
2287	asctime_r()	ftello()	iconv_close()	putchar_unlocked()	2
2288	catclose()	ftw()	iconv_open()	puts()	
2289	catgets()	fwprintf()	ioctl()	pututxline()	
2290	catopen()	fwrite()	link()	putwc()	2
2291	closedir()	fwscanf()	localtime()	putwchar()	2
2292	closelog()	getaddrinfo()	localtime_r()	readdir()	2
2293	ctermid()	getc()	lseek()	readdir_r()	
2294	ctime()	getc_unlocked()	lstat()	remove()	2
2295	ctime_r()	getchar()	mkstemp()	rename()	2
2296	dbm_close()	getchar_unlocked()	mktimes()	rewind()	2
2297	dbm_delete()	getcwd()	nftw()	rewinddir()	
2298	dbm_fetch()	getdate()	opendir()	scanf()	
2299	dbm_nextkey()	getgrent()	openlog()	seekdir()	
2300	dbm_open()	getgrgid()	pathconf()	semop()	2
2301	dbm_store()	getgrgid_r()	pclose()	setgrent()	
2302	dlclose()	getgrnam()	perror()	sethostent()	
2303	dlopen()	getgrnam_r()	popen()	setnetent()	
2304	endgrent()	gethostbyaddr()	posix_fadvise()	setprotoent()	
2305	endhostent()	gethostbyname()	posix_fallocate()	setpwent()	
2306	endnetent()	gethostent()	posix_madvise()	setservent()	
2307	endprotoent()	gethostid()	posix_openpt()	setutxent()	2
2308	endpwent()	gethostname()	posix_spawn()	stat()	2
2309	endservent()	getlogin()	posix_spawnnp()	strerror()	
2310	endutxent()	getlogin_r()	posix_trace_clear()	strerror_r()	2
2311	fclose()	getnameinfo()	posix_trace_close()	strftime()	2
2312	fcntl()†	getnetbyaddr()	posix_trace_create()	symlink()	2
2313	fflush()	getnetbyname()	posix_trace_create_withlog()	sync()	2
2314	fgetc()	getnetent()	posix_trace_eventtypelist_getnext_id()	syslog()	
2315	fgetpos()	getopt()††	posix_trace_eventtypelist_rewind()	tmpfile()	2
2316	fgets()	getprotobyname()	posix_trace_flush()	tmpnam()	
2317	fgetwc()	getprotobynumber()	posix_trace_get_attr()	ttyname()	
2318	fgetws()	getprotoent()	posix_trace_get_filter()	ttyname_r()	
2319	fmtmsg()	getpwent()	posix_trace_get_status()	tzset()	2
2320	fopen()	getpwnam()	posix_trace_getnext_event()	ungetc()	
2321	fpathconf()	getpwnam_r()	posix_trace_open()	ungetwc()	
2322	fprintf()	getpwuid()	posix_trace_rewind()	unlink()	
2323	fputc()	getpwuid_r()	posix_trace_set_filter()	vfprintf()	
2324	fputs()	gets()	posix_trace_shutdown()	vfwprintf()	
2325	fputwc()	getservbyname()	posix_trace_timedgetnext_event()	vprintf()	
2326	fputws()	getservbyport()	posix_TYPED_mem_open()	vwprintf()	
2327	fread()	getservent()	printf()	wcsftime()	2
2328	freopen()	getutxent()	pthread_rwlock_rdlock()	wordexp()	2
2329	fscanf()	getutxid()	pthread_rwlock_timedrdlock()	wprintf()	
2330	fseek()	getutxline()	pthread_rwlock_timedwrlock()	wscanf()	
2331	fseeko()	getwc()	pthread_rwlock_wrlock()		
2332	fsetpos()	getwchar()	putc()		

2333 An implementation shall not introduce cancellation points into any other functions specified in
 2334 this volume of IEEE Std 1003.1-2001.

2335 The side effects of acting upon a cancellation request while suspended during a call of a function
 2336 are the same as the side effects that may be seen in a single-threaded program when a call to a
 2337 function is interrupted by a signal and the given function returns [EINTR]. Any such side effects
 2338 occur before any cancellation cleanup handlers are called.

2339 Whenever a thread has cancelability enabled and a cancellation request has been made with that
 2340 thread as the target, and the thread then calls any function that is a cancellation point (such as
 2341 *pthread_testcancel()* or *read()*), the cancellation request shall be acted upon before the function
 2342 returns. If a thread has cancelability enabled and a cancellation request is made with the thread
 2343 as a target while the thread is suspended at a cancellation point, the thread shall be awakened
 2344 and the cancellation request shall be acted upon. However, if the thread is suspended at a
 2345 cancellation point and the event for which it is waiting occurs before the cancellation request is
 2346 acted upon, it is unspecified whether the cancellation request is acted upon or whether the
 2347 cancellation request remains pending and the thread resumes normal execution.

2348 **2.9.5.3 Thread Cancellation Cleanup Handlers**

2349 Each thread maintains a list of cancellation cleanup handlers. The programmer uses the
 2350 *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions to place routines on and remove
 2351 routines from this list.

2352 When a cancellation request is acted upon, or when a thread calls *pthread_exit()*, the thread first 2
 2353 disables cancellation by setting its cancelability state to PTHREAD_CANCEL_DISABLE and its 2
 2354 cancelability type to PTHREAD_CANCEL_DEFERRED. The cancelability state shall remain set 2
 2355 to PTHREAD_CANCEL_DISABLE until the thread has terminated. The behavior is undefined if 2
 2356 a cancellation cleanup handler or thread-specific data destructor routine changes the 2
 2357 cancelability state to PTHREAD_CANCEL_ENABLE. 2

2358 The routines in the thread's list of cancellation cleanup handlers are invoked one by one in LIFO 2
 2359 sequence; that is, the last routine pushed onto the list (Last In) is the first to be invoked (First 2
 2360 Out). When the cancellation cleanup handler for a scope is invoked, the storage for that scope 2
 2361 remains valid. If the last cancellation cleanup handler returns, thread-specific data destructors (if 2
 2362 any) associated with thread-specific data keys for which the thread has non-NULL values will be 2
 2363 run, in unspecified order, as described for *pthread_key_create()*. 2

2364 After all cancellation cleanup handlers and thread-specific data destructors have returned, 2
 2365 thread execution is terminated. If the thread has terminated because of a call to *pthread_exit()*, 2
 2366 the *value_ptr* argument is made available to any threads joining with the target. If the thread has 2
 2367 terminated by acting on a cancellation request, a status of PTHREAD_CANCELED is made 2
 2368 available to any threads joining with the target. The symbolic constant PTHREAD_CANCELED 2
 2369 expands to a constant expression of type (**void ***) whose value matches no pointer to an object in 2
 2370 memory nor the value NULL. 2

2371 A side effect of acting upon a cancellation request while in a condition variable wait is that the 2
 2372 mutex is re-acquired before calling the first cancellation cleanup handler. In addition, the thread 2
 2373 is no longer considered to be waiting for the condition and the thread shall not have consumed 2
 2374 any pending condition signals on the condition. 2

2375 _____

2376 † For any value of the *cmd* argument.

2377 †† If *opterr* is non-zero.

2378	A cancellation cleanup handler cannot exit via <i>longjmp()</i> or <i>siglongjmp()</i> .
2379	2.9.5.4 Async-Cancel Safety
2380	The <i>pthread_cancel()</i> , <i>pthread_setcancelstate()</i> , and <i>pthread_setcanceltype()</i> functions are defined to
2381	be async-cancel safe.
2382	No other functions in this volume of IEEE Std 1003.1-2001 are required to be async-cancel-safe.
2383	2.9.6 Thread Read-Write Locks
2384	Multiple readers, single writer (read-write) locks allow many threads to have simultaneous
2385	read-only access to data while allowing only one thread to have exclusive write access at any
2386	given time. They are typically used to protect data that is read more frequently than it is
2387	changed.
2388	One or more readers acquire read access to the resource by performing a read lock operation on
2389	the associated read-write lock. A writer acquires exclusive write access by performing a write
2390	lock operation. Basically, all readers exclude any writers and a writer excludes all readers and
2391	any other writers.
2392	A thread that has blocked on a read-write lock (for example, has not yet returned from a
2393	<i>pthread_rwlock_rdlock()</i> or <i>pthread_rwlock_wrlock()</i> call) shall not prevent any unblocked thread
2394	that is eligible to use the same processing resources from eventually making forward progress in
2395	its execution. Eligibility for processing resources shall be determined by the scheduling policy.
2396	Read-write locks can be used to synchronize threads in the current process and other processes if
2397	they are allocated in memory that is writable and shared among the cooperating processes and
2398	have been initialized for this behavior.
2399	2.9.7 Thread Interactions with Regular File Operations
2400	All of the functions <i>chmod()</i> , <i>close()</i> , <i>fchmod()</i> , <i>fcntl()</i> , <i>fstat()</i> , <i>ftruncate()</i> , <i>lseek()</i> , <i>open()</i> , <i>read()</i> ,
2401	<i>readlink()</i> , <i>stat()</i> , <i>symlink()</i> , and <i>write()</i> shall be atomic with respect to each other in the effects
2402	specified in IEEE Std 1003.1-2001 when they operate on regular files. If two threads each call one
2403	of these functions, each call shall either see all of the specified effects of the other call, or none of
2404	them.
2405	2.9.8 Use of Application-Managed Thread Stacks
2406	An “application-managed thread stack” is a region of memory allocated by the application—for
2407	example, memory returned by the <i>malloc()</i> or <i>mmap()</i> functions—and designated as a stack
2408	through the act of passing an address related to that memory as the <i>stackaddr</i> argument to
2409	<i>pthread_attr_setstackaddr()</i> (obsolete) or by passing the address and size of the stack, respectively,
2410	as the <i>stackaddr</i> and <i>stacksize</i> arguments to <i>pthread_attr_setstack()</i> . Application-managed stacks
2411	allow the application to precisely control the placement and size of a stack.
2412	The application grants to the implementation permanent ownership of and control over the
2413	application-managed stack when the attributes object in which the <i>stack</i> or <i>stackaddr</i> attribute has
2414	been set is used, either by presenting that attribute’s object as the <i>attr</i> argument in a call to
2415	<i>pthread_create()</i> that completes successfully, or by storing a pointer to the attributes object in the
2416	<i>sigev_notify_attributes</i> member of a struct sigevent and passing that struct sigevent to a function
2417	accepting such argument that completes successfully. The application may thereafter utilize the
2418	memory within the stack only within the normal context of stack usage within or properly
2419	synchronized with a thread that has been scheduled by the implementation with stack pointer
2420	value(s) that are within the range of that stack. In particular, the region of memory cannot be

2421	freed, nor can it be later specified as the stack for another thread.	2
2422	When specifying an attributes object with an application-managed stack through the	2
2423	<i>sigev_notify_attributes</i> member of a struct sigevent , the results are undefined if the requested	2
2424	signal is generated multiple times (as for a repeating timer).	2
2425	Until an attributes object in which the <i>stack</i> or <i>stackaddr</i> attribute has been set is used, the	2
2426	application retains ownership of and control over the memory allocated to the stack. It may free	2
2427	or reuse the memory as long as it either deletes the attributes object, or before using the	2
2428	attributes object replaces the stack by making an additional call to the same function, either	2
2429	<i>pthread_attr_setstackaddr()</i> or <i>pthread_attr_setstack()</i> , that was used originally to designate the	2
2430	stack. There is no mechanism to retract the reference to an application-managed stack by an	2
2431	existing attributes object.	2
2432	Once an attributes object with an application-managed stack has been used, that attributes	2
2433	object cannot be used again by a subsequent call to <i>pthread_create()</i> or any function accepting a	2
2434	struct sigevent with <i>sigev_notify_attributes</i> containing a pointer to the attributes object, without	2
2435	designating an unused application-managed stack by making an additional call to the function	2
2436	originally used to define the stack, <i>pthread_attr_setstack()</i> or <i>pthread_attr_setstackaddr()</i> .	2

2437 2.10 Sockets

2438 A socket is an endpoint for communication using the facilities described in this section. A socket
 2439 is created with a specific socket type, described in Section 2.10.6 (on page 60), and is associated
 2440 with a specific protocol, detailed in Section 2.10.3 (on page 60). A socket is accessed via a file
 2441 descriptor obtained when the socket is created.

2442 2.10.1 Address Families

2443 All network protocols are associated with a specific address family. An address family provides
 2444 basic services to the protocol implementation to allow it to function within a specific network
 2445 environment. These services may include packet fragmentation and reassembly, routing,
 2446 addressing, and basic transport. An address family is normally comprised of a number of
 2447 protocols, one per socket type. Each protocol is characterized by an abstract socket type. It is not
 2448 required that an address family support all socket types. An address family may contain
 2449 multiple protocols supporting the same socket abstraction.

2450 Section 2.10.17 (on page 67), Section 2.10.19 (on page 68), and Section 2.10.20 (on page 68),
 2451 respectively, describe the use of sockets for local UNIX connections, for Internet protocols based
 2452 on IPv4, and for Internet protocols based on IPv6.

2453 2.10.2 Addressing

2454 An address family defines the format of a socket address. All network addresses are described
 2455 using a general structure, called a **sockaddr**, as defined in the Base Definitions volume of
 2456 IEEE Std 1003.1-2001, <sys/socket.h>. However, each address family imposes finer and more
 2457 specific structure, generally defining a structure with fields specific to the address family. The
 2458 field *sa_family* in the **sockaddr** structure contains the address family identifier, specifying the
 2459 format of the *sa_data* area. The size of the *sa_data* area is unspecified.

2.10.3 Protocols

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

2.10.4 Routing

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

2.10.5 Interfaces

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

2.10.6 Socket Types

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

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

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

The `SOCK_DGRAM` socket type supports connectionless data transfer which is not necessarily acknowledged or reliable. Datagrams may be sent to the address specified (possibly multicast or broadcast) in each output operation, and incoming datagrams may be received from multiple sources. The source address of each datagram is available when receiving the datagram. An application may also pre-specify a peer address, in which case calls to output functions shall send to the pre-specified peer. If a peer has been specified, only datagrams from that peer shall be received. A datagram must be sent in a single output operation, and must be received in a single input operation. The maximum size of a datagram is protocol-specific; with some protocols, the limit is implementation-defined. Output datagrams may be buffered within the system; thus, a successful return from an output function does not guarantee that a datagram is

2505 actually sent or received. However, implementations should attempt to detect any errors
2506 possible before the return of an output function, reporting any error by an unsuccessful return
2507 value.

2508 RS The SOCK_RAW socket type is similar to the SOCK_DGRAM type. It differs in that it is
2509 normally used with communication providers that underlie those used for the other socket
2510 types. For this reason, the creation of a socket with type SOCK_RAW shall require appropriate
2511 privilege. The format of datagrams sent and received with this socket type generally include
2512 specific protocol headers, and the formats are protocol-specific and implementation-defined.

2513 2.10.7 Socket I/O Mode

2514 The I/O mode of a socket is described by the O_NONBLOCK file status flag which pertains to
2515 the open file description for the socket. This flag is initially off when a socket is created, but may
2516 be set and cleared by the use of the F_SETFL command of the *fctl()* function.

2517 When the O_NONBLOCK flag is set, functions that would normally block until they are
2518 complete shall either return immediately with an error, or shall complete asynchronously to the
2519 execution of the calling process. Data transfer operations (the *read()*, *write()*, *send()*, and *recv()*
2520 functions) shall complete immediately, transfer only as much as is available, and then return
2521 without blocking, or return an error indicating that no transfer could be made without blocking.
2522 The *connect()* function initiates a connection and shall return without blocking when
2523 O_NONBLOCK is set; it shall return the error [EINPROGRESS] to indicate that the connection
2524 was initiated successfully, but that it has not yet completed.

2525 2.10.8 Socket Owner

2526 The owner of a socket is unset when a socket is created. The owner may be set to a process ID or
2527 process group ID using the F_SETOWN command of the *fctl()* function.

2528 2.10.9 Socket Queue Limits

2529 The transmit and receive queue sizes for a socket are set when the socket is created. The default
2530 sizes used are both protocol-specific and implementation-defined. The sizes may be changed
2531 using the *setssockopt()* function.

2532 2.10.10 Pending Error

2533 Errors may occur asynchronously, and be reported to the socket in response to input from the
2534 network protocol. The socket stores the pending error to be reported to a user of the socket at the
2535 next opportunity. The error is returned in response to a subsequent *send()*, *recv()*, or *getsockopt()*
2536 operation on the socket, and the pending error is then cleared.

2537 2.10.11 Socket Receive Queue

2538 A socket has a receive queue that buffers data when it is received by the system until it is
2539 removed by a receive call. Depending on the type of the socket and the communication provider,
2540 the receive queue may also contain ancillary data such as the addressing and other protocol data
2541 associated with the normal data in the queue, and may contain out-of-band or expedited data.
2542 The limit on the queue size includes any normal, out-of-band data, datagram source addresses,
2543 and ancillary data in the queue. The description in this section applies to all sockets, even though
2544 some elements cannot be present in some instances.

2545 The contents of a receive buffer are logically structured as a series of data segments with
2546 associated ancillary data and other information. A data segment may contain normal data or

2547 out-of-band data, but never both. A data segment may complete a record if the protocol
2548 supports records (always true for types SOCK_SEQPACKET and SOCK_DGRAM). A record
2549 may be stored as more than one segment; the complete record might never be present in the
2550 receive buffer at one time, as a portion might already have been returned to the application, and
2551 another portion might not yet have been received from the communications provider. A data
2552 segment may contain ancillary protocol data, which is logically associated with the segment.
2553 Ancillary data is received as if it were queued along with the first normal data octet in the
2554 segment (if any). A segment may contain ancillary data only, with no normal or out-of-band
2555 data. For the purposes of this section, a datagram is considered to be a data segment that
2556 terminates a record, and that includes a source address as a special type of ancillary data. Data
2557 segments are placed into the queue as data is delivered to the socket by the protocol. Normal
2558 data segments are placed at the end of the queue as they are delivered. If a new segment
2559 contains the same type of data as the preceding segment and includes no ancillary data, and if
2560 the preceding segment does not terminate a record, the segments are logically merged into a
2561 single segment.

2562 The receive queue is logically terminated if an end-of-file indication has been received or a
2563 connection has been terminated. A segment shall be considered to be terminated if another
2564 segment follows it in the queue, if the segment completes a record, or if an end-of-file or other
2565 connection termination has been reported. The last segment in the receive queue shall also be
2566 considered to be terminated while the socket has a pending error to be reported.

2567 A receive operation shall never return data or ancillary data from more than one segment.

2568 2.10.12 Socket Out-of-Band Data State

2569 The handling of received out-of-band data is protocol-specific. Out-of-band data may be placed
2570 in the socket receive queue, either at the end of the queue or before all normal data in the queue.
2571 In this case, out-of-band data is returned to an application program by a normal receive call.
2572 Out-of-band data may also be queued separately rather than being placed in the socket receive
2573 queue, in which case it shall be returned only in response to a receive call that requests out-of-
2574 band data. It is protocol-specific whether an out-of-band data mark is placed in the receive
2575 queue to demarcate data preceding the out-of-band data and following the out-of-band data. An
2576 out-of-band data mark is logically an empty data segment that cannot be merged with other
2577 segments in the queue. An out-of-band data mark is never returned in response to an input
2578 operation. The *socketmark()* function can be used to test whether an out-of-band data mark is the
2579 first element in the queue. If an out-of-band data mark is the first element in the queue when an
2580 input function is called without the MSG_PEEK option, the mark is removed from the queue and
2581 the following data (if any) is processed as if the mark had not been present.

2582 2.10.13 Connection Indication Queue

2583 Sockets that are used to accept incoming connections maintain a queue of outstanding
2584 connection indications. This queue is a list of connections that are awaiting acceptance by the
2585 application; see *listen()*.

2586 2.10.14 Signals

2587 One category of event at the socket interface is the generation of signals. These signals report
2588 protocol events or process errors relating to the state of the socket. The generation or delivery of
2589 a signal does not change the state of the socket, although the generation of the signal may have
2590 been caused by a state change.

2591 The SIGPIPE signal shall be sent to a thread that attempts to send data on a socket that is no
2592 longer able to send. In addition, the send operation fails with the error [EPIPE].

2593 If a socket has an owner, the SIGURG signal is sent to the owner of the socket when it is notified
2594 of expedited or out-of-band data. The socket state at this time is protocol-dependent, and the
2595 status of the socket is specified in Section 2.10.17 (on page 67), Section 2.10.19 (on page 68), and
2596 Section 2.10.20 (on page 68). Depending on the protocol, the expedited data may or may not
2597 have arrived at the time of signal generation.

2598 2.10.15 Asynchronous Errors

2599 If any of the following conditions occur asynchronously for a socket, the corresponding value
2600 listed below shall become the pending error for the socket:

2601 [ECONNABORTED]

2602 The connection was aborted locally.

2603 [ECONNREFUSED]

2604 For a connection-mode socket attempting a non-blocking connection, the attempt to connect
2605 was forcefully rejected. For a connectionless-mode socket, an attempt to deliver a datagram
2606 was forcefully rejected.

2607 [ECONNRESET]

2608 The peer has aborted the connection.

2609 [EHOSTDOWN]

2610 The destination host has been determined to be down or disconnected.

2611 [EHOSTUNREACH]

2612 The destination host is not reachable.

2613 [EMSGSIZE]

2614 For a connectionless-mode socket, the size of a previously sent datagram prevented
2615 delivery.

2616 [ENETDOWN]

2617 The local network connection is not operational.

2618 [ENETRESET]

2619 The connection was aborted by the network.

2620 [ENETUNREACH]

2621 The destination network is not reachable.

2.10.16 Use of Options

2623 There are a number of socket options which either specialize the behavior of a socket or provide
2624 useful information. These options may be set at different protocol levels and are always present
2625 at the uppermost “socket” level.

2626 Socket options are manipulated by two functions, *getsockopt()* and *setsockopt()*. These functions
2627 allow an application program to customize the behavior and characteristics of a socket to
2628 provide the desired effect.

2629 All of the options have default values. The type and meaning of these values is defined by the
2630 protocol level to which they apply. Instead of using the default values, an application program
2631 may choose to customize one or more of the options. However, in the bulk of cases, the default
2632 values are sufficient for the application.

2633 Some of the options are used to enable or disable certain behavior within the protocol modules
2634 (for example, turn on debugging) while others may be used to set protocol-specific information
2635 (for example, IP time-to-live on all the application’s outgoing packets). As each of the options is
2636 introduced, its effect on the underlying protocol modules is described.

2637 Table 2-1 shows the value for the socket level.

2638 **Table 2-1** Value of Level for Socket Options

Name	Description
SOL_SOCKET	Options are intended for the sockets level.

2641 Table 2-2 (on page 65) lists those options present at the socket level; that is, when the *level*
2642 parameter of the *getsockopt()* or *setsockopt()* function is SOL_SOCKET, the types of the option
2643 value parameters associated with each option, and a brief synopsis of the meaning of the option
2644 value parameter. Unless otherwise noted, each may be examined with *getsockopt()* and set with
2645 *setsockopt()* on all types of socket.

2646

2647

2648

2649

2650

2651

2652

2653

2654

2655

2656

2657

2658

2659

2660

2661

2662

2663

2664

2665

2666

2667

2668

2669

2670

2671

2672

2673

2674

2675

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.

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

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

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

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

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

The SO_RCVLOWAT option sets the minimum number of bytes to process for socket input operations. In general, receive calls block until any (non-zero) amount of data is received, then return the smaller of the amount available or the amount requested. The default value for SO_RCVLOWAT is 1, and does not affect the general case. 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. Receive calls may still 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). As mentioned previously, the default value for SO_RCVLOWAT is 1 byte. It is implementation-defined whether the SO_RCVLOWAT option can be set.

The SO_RCVTIMEO option is an option to set a timeout value for input operations. 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 *errno* shall be set to [EWOULDBLOCK] if no data were received. The default for this option is the value zero, which indicates that a receive operation will not time out. It is implementation-defined whether the SO_RCVTIMEO option can be set.

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

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

The SO_SNDLOWAT option sets the minimum number of bytes to process for socket output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Non-blocking output 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_SNDSNTOIMEO 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 65), 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_SNDSNTOIMEO 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.

RS SOCK_SEQPACKET Unspecified.

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.

2784 2.10.19 Use of Sockets over Internet Protocols Based on IPv4

2785 Support for sockets over Internet protocols based on IPv4 is mandatory.

2786 2.10.19.1 Headers

2787 The symbolic constant AF_INET defined in the `<sys/socket.h>` header is used to identify the
2788 IPv4 Internet address family. The `<netinet/in.h>` header contains other definitions used in
2789 connection with IPv4 Internet sockets. See the Base Definitions volume of IEEE Std 1003.1-2001,
2790 Chapter 13, Headers.

2791 The `sockaddr_storage` structure defined in `<sys/socket.h>` shall be large enough to
2792 accommodate a `sockaddr_in` structure (see the `<netinet/in.h>` header defined in the Base
2793 Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers) and shall be aligned at an
2794 appropriate boundary so that pointers to it can be cast as pointers to `sockaddr_in` structures and
2795 used to access the fields of those structures without alignment problems. When a
2796 `sockaddr_storage` structure is cast as a `sockaddr_in` structure, the `ss_family` field maps onto the
2797 `sin_family` field.

2798 2.10.20 Use of Sockets over Internet Protocols Based on IPv6

2799 IP6 This section describes extensions to support sockets over Internet protocols based on IPv6. This
2800 functionality is dependent on support of the IPV6 option (and the rest of this section is not
2801 further shaded for this option).

2802 To enable smooth transition from IPv4 to IPv6, the features defined in this section may, in certain
2803 circumstances, also be used in connection with IPv4; see Section 2.10.20.2 (on page 69).

2804 2.10.20.1 Addressing

2805 IPv6 overcomes the addressing limitations of previous versions by using 128-bit addresses
2806 instead of 32-bit addresses. The IPv6 address architecture is described in RFC 2373.

2807 There are three kinds of IPv6 address:

2808 Unicast

2809 Identifies a single interface.

2810 A unicast address can be global, link-local (designed for use on a single link), or site-local
2811 (designed for systems not connected to the Internet). Link-local and site-local addresses
2812 need not be globally unique.

2813 Anycast

2814 Identifies a set of interfaces such that a packet sent to the address can be delivered to any
2815 member of the set.

2816 An anycast address is similar to a unicast address; the nodes to which an anycast address is
2817 assigned must be explicitly configured to know that it is an anycast address.

2818 Multicast

2819 Identifies a set of interfaces such that a packet sent to the address should be delivered to
2820 every member of the set.

2821 An application can send multicast datagrams by simply specifying an IPv6 multicast
2822 address in the `address` argument of `sendto()`. To receive multicast datagrams, an application
2823 must join the multicast group (using `setsockopt()` with `IPV6_JOIN_GROUP`) and must bind
2824 to the socket the UDP port on which datagrams will be received. Some applications should
2825 also bind the multicast group address to the socket, to prevent other datagrams destined to
2826 that port from being delivered to the socket.

2827 A multicast address can be global, node-local, link-local, site-local, or organization-local.

2828 The following special IPv6 addresses are defined:

2829 Unspecified

2830 An address that is not assigned to any interface and is used to indicate the absence of an
2831 address.

2832 Loopback

2833 A unicast address that is not assigned to any interface and can be used by a node to send
2834 packets to itself.

2835 Two sets of IPv6 addresses are defined to correspond to IPv4 addresses:

2836 IPv4-compatible addresses

2837 These are assigned to nodes that support IPv6 and can be used when traffic is “tunneled”
2838 through IPv4.

2839 IPv4-mapped addresses

2840 These are used to represent IPv4 addresses in IPv6 address format; see Section 2.10.20.2.

2841 Note that the unspecified address and the loopback address must not be treated as IPv4-
2842 compatible addresses.

2843 2.10.20.2 Compatibility with IPv4

2844 The API provides the ability for IPv6 applications to interoperate with applications using IPv4,
2845 by using IPv4-mapped IPv6 addresses. These addresses can be generated automatically by the
2846 *getaddrinfo()* function when the specified host has only IPv4 addresses.

2847 Applications can use AF_INET6 sockets to open TCP connections to IPv4 nodes, or send UDP
2848 packets to IPv4 nodes, by simply encoding the destination's IPv4 address as an IPv4-mapped
2849 IPv6 address, and passing that address, within a **sockaddr_in6** structure, in the *connect()*,
2850 *sendto()*, or *sendmsg()* function. When applications use AF_INET6 sockets to accept TCP
2851 connections from IPv4 nodes, or receive UDP packets from IPv4 nodes, the system shall return
2852 the peer's address to the application in the *accept()*, *recvfrom()*, *recvmsg()*, or *getpeername()*
2853 function using a **sockaddr_in6** structure encoded this way. If a node has an IPv4 address, then
2854 the implementation shall allow applications to communicate using that address via an
2855 AF_INET6 socket. In such a case, the address will be represented at the API by the
2856 corresponding IPv4-mapped IPv6 address. Also, the implementation may allow an AF_INET6
2857 socket bound to **in6addr_any** to receive inbound connections and packets destined to one of the
2858 node's IPv4 addresses.

2859 An application can use AF_INET6 sockets to bind to a node's IPv4 address by specifying the
2860 address as an IPv4-mapped IPv6 address in a **sockaddr_in6** structure in the *bind()* function. For
2861 an AF_INET6 socket bound to a node's IPv4 address, the system shall return the address in the
2862 *getsockname()* function as an IPv4-mapped IPv6 address in a **sockaddr_in6** structure.

2863 2.10.20.3 Interface Identification

2864 Each local interface is assigned a unique positive integer as a numeric index. Indexes start at 1;
2865 zero is not used. There may be gaps so that there is no current interface for a particular positive
2866 index. Each interface also has a unique implementation-defined name.

2867 2.10.20.4 Options

2868 The following options apply at the IPPROTO_IPV6 level:

2869 IPV6_JOIN_GROUP

2870 When set via *setsockopt()*, it joins the application to a multicast group on an interface
2871 (identified by its index) and addressed by a given multicast address, enabling packets sent
2872 to that address to be read via the socket. If the interface index is specified as zero, the
2873 system selects the interface (for example, by looking up the address in a routing table and
2874 using the resulting interface).

2875 An attempt to read this option using *getsockopt()* shall result in an [EOPNOTSUPP] error.

2876 The parameter type of this option is a pointer to an **ipv6_mreq** structure.

2877 IPV6_LEAVE_GROUP

2878 When set via *setsockopt()*, it removes the application from the multicast group on an
2879 interface (identified by its index) and addressed by a given multicast address.

2880 An attempt to read this option using *getsockopt()* shall result in an [EOPNOTSUPP] error.

2881 The parameter type of this option is a pointer to an **ipv6_mreq** structure.

2882 IPV6_MULTICAST_HOPS

2883 The value of this option is the hop limit for outgoing multicast IPv6 packets sent via the
2884 socket. Its possible values are the same as those of IPV6_UNICAST_HOPS. If the
2885 IPV6_MULTICAST_HOPS option is not set, a value of 1 is assumed. This option can be set
2886 via *setsockopt()* and read via *getsockopt()*.

2887 The parameter type of this option is a pointer to an **int**. (Default value: 1)

2888 IPV6_MULTICAST_IF

2889 The index of the interface to be used for outgoing multicast packets. It can be set via
2890 *setsockopt()* and read via *getsockopt()*. If the interface index is specified as zero, the system
2891 selects the interface (for example, by looking up the address in a routing table and using the
2892 resulting interface).

2893 The parameter type of this option is a pointer to an **unsigned int**. (Default value: 0)

2894 IPV6_MULTICAST_LOOP

2895 This option controls whether outgoing multicast packets should be delivered back to the
2896 local application when the sending interface is itself a member of the destination multicast
2897 group. If it is set to 1 they are delivered. If it is set to 0 they are not. Other values result in an
2898 [EINVAL] error. This option can be set via *setsockopt()* and read via *getsockopt()*.

2899 The parameter type of this option is a pointer to an **unsigned int** which is used as a Boolean
2900 value. (Default value: 1)

2901 IPV6_UNICAST_HOPS

2902 The value of this option is the hop limit for outgoing unicast IPv6 packets sent via the
2903 socket. If the option is not set, or is set to -1, the system selects a default value. Attempts to
2904 set a value less than -1 or greater than 255 shall result in an [EINVAL] error. This option can
2905 be set via *setsockopt()* and read via *getsockopt()*.

2906 The parameter type of this option is a pointer to an **int**. (Default value: Unspecified)

2907 IPV6_V6ONLY

2908 This socket option restricts AF_INET6 sockets to IPv6 communications only. AF_INET6
2909 sockets may be used for both IPv4 and IPv6 communications. Some applications may want
2910 to restrict their use of an AF_INET6 socket to IPv6 communications only. For these

2911 applications, the IPV6_V6ONLY socket option is defined. When this option is turned on, the
 2912 socket can be used to send and receive IPv6 packets only. This is an IPPROTO_IPV6-level
 2913 option.

2914 The parameter type of this option is a pointer to an **int** which is used as a Boolean value.
 2915 (Default value: 0)

2916 An [EOPNOTSUPP] error shall result if IPV6_JOIN_GROUP or IPV6_LEAVE_GROUP is used
 2917 with *getsockopt()*.

2918 2.10.20.5 Headers

2919 The symbolic constant AF_INET6 is defined in the **<sys/socket.h>** header to identify the IPv6
 2920 Internet address family. See the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13,
 2921 Headers.

2922 The **sockaddr_storage** structure defined in **<sys/socket.h>** shall be large enough to
 2923 accommodate a **sockaddr_in6** structure (see the **<netinet/in.h>** header defined in the Base
 2924 Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers) and shall be aligned at an
 2925 appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_in6** structures
 2926 and used to access the fields of those structures without alignment problems. When a
 2927 **sockaddr_storage** structure is cast as a **sockaddr_in6** structure, the *ss_family* field maps onto the
 2928 *sin6_family* field.

2929 The **<netinet/in.h>**, **<arpa/inet.h>**, and **<netdb.h>** headers contain other definitions used in
 2930 connection with IPv6 Internet sockets; see the Base Definitions volume of IEEE Std 1003.1-2001,
 2931 Chapter 13, Headers.

2932 2.11 Tracing

2933 TRC This section describes extensions to support tracing of user applications. This functionality is
 2934 dependent on support of the Trace option (and the rest of this section is not further shaded for
 2935 this option).

2936 The tracing facilities defined in IEEE Std 1003.1-2001 allow a process to select a set of trace event
 2937 types, to activate a trace stream of the selected trace events as they occur in the flow of
 2938 execution, and to retrieve the recorded trace events.

2939 The tracing operation relies on three logically different components: the traced process, the
 2940 controller process, and the analyzer process. During the execution of the traced process, when a
 2941 trace point is reached, a trace event is recorded into the trace streams created for that process in
 2942 which the associated trace event type identifier is not being filtered out. The controller process
 2943 controls the operation of recording the trace events into the trace stream. It shall be able to:

- 2944 • Initialize the attributes of a trace stream
- 2945 • Create the trace stream (for a specified traced process) using those attributes
- 2946 • Start and stop tracing for the trace stream
- 2947 • Filter the type of trace events to be recorded, if the Trace Event Filter option is supported
- 2948 • Shut a trace stream down

2949 These operations can be done for an active trace stream. The analyzer process retrieves the
 2950 traced events either at runtime, when the trace stream has not yet been shut down, but is still
 2951 recording trace events; or after opening a trace log that had been previously recorded and shut
 2952 down. These three logically different operations can be performed by the same process, or can be

2953 distributed into different processes.

2954 A trace stream identifier can be created by a call to *posix_trace_create()*,
2955 *posix_trace_create_withlog()*, or *posix_trace_open()*. The *posix_trace_create()* and
2956 *posix_trace_create_withlog()* functions should be used by a controller process. The
2957 *posix_trace_open()* should be used by an analyzer process.

2958 The tracing functions can serve different purposes. One purpose is debugging the possibly pre-
2959 instrumented code, while another is post-mortem fault analysis. These two potential uses differ
2960 in that the first requires pre-filtering capabilities to avoid overwhelming the trace stream and
2961 permits focusing on expected information; while the second needs comprehensive trace
2962 capabilities in order to be able to record all types of information.

2963 The events to be traced belong to two classes:

- 2964 1. User trace events (generated by the application instrumentation)
- 2965 2. System trace events (generated by the operating system)

2966 The trace interface defines several system trace event types associated with control of and
2967 operation of the trace stream. This small set of system trace events includes the minimum
2968 required to interpret correctly the trace event information present in the stream. Other desirable
2969 system trace events for some particular application profile may be implemented and are
2970 encouraged; for example, process and thread scheduling, signal occurrence, and so on.

2971 Each traced process shall have a mapping of the trace event names to trace event type identifiers
2972 that have been defined for that process. Each active trace stream shall have a mapping that
2973 incorporates all the trace event type identifiers predefined by the trace system plus all the
2974 mappings of trace event names to trace event type identifiers of the processes that are being
2975 traced into that trace stream. These mappings are defined from the instrumented application by
2976 calling the *posix_trace_eventid_open()* function and from the controller process by calling the
2977 *posix_trace_trid_eventid_open()* function. For a pre-recorded trace stream, the list of trace event
2978 types is obtained from the pre-recorded trace log.

2979 The *st_ctime* and *st_mtime* fields of a file associated with an active trace stream shall be marked
2980 for update every time any of the tracing operations modifies that file.

2981 The *st_atime* field of a file associated with a trace stream shall be marked for update every time
2982 any of the tracing operations causes data to be read from that file.

2983 Results are undefined if the application performs any operation on a file descriptor associated
2984 with an active or pre-recorded trace stream until *posix_trace_shutdown()* or *posix_trace_close()* is
2985 called for that trace stream.

2986 The main purpose of this option is to define a complete set of functions and concepts that allow
2987 a conforming application to be traced from creation to termination, whatever its realtime
2988 constraints and properties.

2989 2.11.1 Tracing Data Definitions

2990 2.11.1.1 Structures

2991 The <*trace.h*> header shall define the *posix_trace_status_info* and *posix_trace_event_info* structures
2992 described below. Implementations may add extensions to these structures.

2993 **posix_trace_status_info Structure**

2994 To facilitate control of a trace stream, information about the current state of an active trace
 2995 stream can be obtained dynamically. This structure is returned by a call to the
 2996 **posix_trace_get_status()** function.

2997 The **posix_trace_status_info** structure defined in <trace.h> shall contain at least the following
 2998 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_overrun_status</i>	Indicates whether trace events were lost in the trace stream.

3005 If the Trace Log option is supported in addition to the Trace option, the **posix_trace_status_info**
 3006 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_overrun_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.

3015 The *posix_stream_status* member indicates the operating mode of the trace stream and shall have
 3016 one of the following values defined by manifest constants in the <trace.h> header:

3017 **POSIX_TRACE_RUNNING**

3018 Tracing is in progress; that is, the trace stream is accepting trace events.

3019 **POSIX_TRACE_SUSPENDED**

3020 The trace stream is not accepting trace events. The tracing operation has not yet started or
 3021 has stopped, either following a *posix_trace_stop()* function call or because the trace resources
 3022 are exhausted.

3023 The *posix_stream_full_status* member indicates the full status of the trace stream, and it shall have
 3024 one of the following values defined by manifest constants in the <trace.h> header:

3025 **POSIX_TRACE_FULL**

3026 The space in the trace stream for trace events is exhausted.

3027 **POSIX_TRACE_NOT_FULL**

3028 There is still space available in the trace stream.

3029 The combination of the *posix_stream_status* and *posix_stream_full_status* members also indicates
 3030 the actual status of the stream. The status shall be interpreted as follows:

3031 **POSIX_TRACE_RUNNING and POSIX_TRACE_NOT_FULL**

3032 This status combination indicates that tracing is in progress, and there is space available for
 3033 recording more trace events.

3034 **POSIX_TRACE_RUNNING and POSIX_TRACE_FULL**

3035 This status combination indicates that tracing is in progress and that the trace stream is full
 3036 of trace events. This status combination cannot occur unless the *stream-full-policy* is set to

3037 POSIX_TRACE_LOOP. The trace stream contains trace events recorded during a moving
3038 time window of prior trace events, and some older trace events may have been overwritten
3039 and thus lost.

3040 **POSIX_TRACE_SUSPENDED** and **POSIX_TRACE_NOT_FULL**

3041 This status combination indicates that tracing has not yet been started, has been stopped by
3042 the *posix_trace_stop()* function, or has been cleared by the *posix_trace_clear()* function.

3043 **POSIX_TRACE_SUSPENDED** and **POSIX_TRACE_FULL**

3044 This status combination indicates that tracing has been stopped by the implementation
3045 because the *stream-full-policy* attribute was **POSIX_TRACE_UNTIL_FULL** and trace
3046 resources were exhausted, or that the trace stream was stopped by the function
3047 *posix_trace_stop()* at a time when trace resources were exhausted.

3048 The *posix_stream_overrun_status* member indicates whether trace events were lost in the trace
3049 stream, and shall have one of the following values defined by manifest constants in the
3050 **<trace.h>** header:

3051 **POSIX_TRACE_OVERRUN**

3052 At least one trace event was lost and thus was not recorded in the trace stream.

3053 **POSIX_TRACE_NO_OVERRUN**

3054 No trace events were lost.

3055 When the corresponding trace stream is created, the *posix_stream_overrun_status* member shall be
3056 set to **POSIX_TRACE_NO_OVERRUN**.

3057 Whenever an overrun occurs, the *posix_stream_overrun_status* member shall be set to
3058 **POSIX_TRACE_OVERRUN**.

3059 An overrun occurs when:

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

3065 The *posix_stream_overrun_status* member is reset to zero after its value is read.

3066 If the Trace Log option is supported in addition to the Trace option, the *posix_stream_flush_status*,
3067 *posix_stream_flush_error*, *posix_log_overrun_status*, and *posix_log_full_status* members are defined
3068 as follows; otherwise, they are undefined.

3069 The *posix_stream_flush_status* member indicates whether a flush operation is being performed
3070 and shall have one of the following values defined by manifest constants in the header
3071 **<trace.h>**:

3072 **POSIX_TRACE_FLUSHING**

3073 The trace stream is currently being flushed to the trace log.

3074 **POSIX_TRACE_NOT_FLUSHING**

3075 No flush operation is in progress.

3076 The *posix_stream_flush_status* member shall be set to **POSIX_TRACE_FLUSHING** if a flush
3077 operation is in progress either due to a call to the *posix_trace_flush()* function (explicit or caused
3078 by a trace stream shutdown operation) or because the trace stream has become full with the
3079 *stream-full-policy* attribute set to **POSIX_TRACE_FLUSH**. The *posix_stream_flush_status* member
3080 shall be set to **POSIX_TRACE_NOT_FLUSHING** if no flush operation is in progress.

3081 The *posix_stream_flush_error* member shall be set to zero if no error occurred during flushing. If
 3082 an error occurred during a previous flushing operation, the *posix_stream_flush_error* member
 3083 shall be set to the value of the first error that occurred. If more than one error occurs while
 3084 flushing, error values after the first shall be discarded. The *posix_stream_flush_error* member is
 3085 reset to zero after its value is read.

3086 The *posix_log_overrun_status* member indicates whether trace events were lost in the trace log,
 3087 and shall have one of the following values defined by manifest constants in the <trace.h>
 3088 header:

3089 **POSIX_TRACE_OVERRUN**
 3090 At least one trace event was lost.

3091 **POSIX_TRACE_NO_OVERRUN**
 3092 No trace events were lost.

3093 When the corresponding trace stream is created, the *posix_log_overrun_status* member shall be set
 3094 to **POSIX_TRACE_NO_OVERRUN**. Whenever an overrun occurs, this status shall be set to
 3095 **POSIX_TRACE_OVERRUN**. The *posix_log_overrun_status* member is reset to zero after its value
 3096 is read.

3097 The *posix_log_full_status* member indicates the full status of the trace log, and it shall have one of
 3098 the following values defined by manifest constants in the <trace.h> header:

3099 **POSIX_TRACE_FULL**
 3100 The space in the trace log is exhausted.

3101 **POSIX_TRACE_NOT_FULL**
 3102 There is still space available in the trace log.

3103 The *posix_log_full_status* member is only meaningful if the *log-full-policy* attribute is either
 3104 **POSIX_TRACE_UNTIL_FULL** or **POSIX_TRACE_LOOP**.

3105 For an active trace stream without log, that is created by the *posix_trace_create()* function, the
 3106 *posix_log_overrun_status* member shall be set to **POSIX_TRACE_NO_OVERRUN** and the
 3107 *posix_log_full_status* member shall be set to **POSIX_TRACE_NOT_FULL**.

3108 **posix_trace_event_info** Structure

3109 The trace event structure **posix_trace_event_info** contains the information for one recorded
 3110 trace event. This structure is returned by the set of functions *posix_trace_getnext_event()*,
 3111 *posix_trace_timedgetnext_event()*, and *posix_trace_trygetnext_event()*.

3112 The **posix_trace_event_info** structure defined in <trace.h> shall contain at least the following
 3113 members:

3114 Member Type	3115 Member Name	3116 Description
3115 trace_event_id_t 3116 pid_t	3116 <i>posix_event_id</i> 3117 <i>posix_pid</i>	Trace event type identification. Process ID of the process that generated the trace event.
3118 void * 3119 int	3118 <i>posix_prog_address</i> 3119 <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.
3120 struct timespec	3121 <i>posix_timestamp</i>	Time at which the trace event was generated.

3122 In addition, if the Trace option and the Threads option are both supported, the
 3123 **posix_trace_event_info** structure defined in <trace.h> shall contain the following additional
 3124 member:

3125
3126
3127
3128
3129

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

3130 The `posix_event_id` member represents the identification of the trace event type and its value is
 3131 not directly defined by the user. This identification is returned by a call to one of the following
 3132 functions: `posix_trace_trid_eventid_open()`, `posix_trace_eventtypelist_getnext_id()`, or
 3133 `posix_trace_eventid_open()`. The name of the trace event type can be obtained by calling
 3134 `posix_trace_eventid_get_name()`.

3135 The `posix_pid` is the process identifier of the traced process which generated the trace event. If
 3136 the `posix_event_id` member is one of the implementation-defined system trace events and that
 3137 trace event is not associated with any process, the `posix_pid` member shall be set to zero.

3138 For a user trace event, the `posix_prog_address` member is the process mapped address of the point
 3139 at which the associated call to the `posix_trace_event()` function was made. For a system trace
 3140 event, if the trace event is caused by a system service explicitly called by the application, the
 3141 `posix_prog_address` member shall be the address of the process at the point where the call to that
 3142 system service was made.

3143 The `posix_truncation_status` member defines whether the data associated with a trace event has
 3144 been truncated at the time the trace event was generated, or at the time the trace event was read
 3145 from the trace stream, or (if the Trace Log option is supported) from the trace log (see the `event`
 3146 argument from the `posix_trace_getnext_event()` function). The `posix_truncation_status` member
 3147 shall have one of the following values defined by manifest constants in the <trace.h> header:

3148 POSIX_TRACE_NOT_TRUNCATED

3149 All the traced data is available.

3150 POSIX_TRACE_TRUNCATED_RECORD

3151 Data was truncated at the time the trace event was generated.

3152 POSIX_TRACE_TRUNCATED_READ

3153 Data was truncated at the time the trace event was read from a trace stream or a trace log
 3154 because the reader's buffer was too small. This truncation status overrides the
 3155 POSIX_TRACE_TRUNCATED_RECORD status.

3156 The `posix_timestamp` member shall be the time at which the trace event was generated. The clock
 3157 used is implementation-defined, but the resolution of this clock can be retrieved by a call to the
 3158 `posix_trace_attr_getclockres()` function.

3159 If the Threads option is supported in addition to the Trace option:

- The `posix_thread_id` member is the identifier of the thread that generated the trace event. If
 3161 the `posix_event_id` member is one of the implementation-defined system trace events and that
 3162 trace event is not associated with any thread, the `posix_thread_id` member shall be set to zero.

3163 Otherwise, this member is undefined.

3164 2.11.1.2 Trace Stream Attributes

3165 Trace streams have attributes that compose the `posix_trace_attr_t` trace stream attributes object.
 3166 This object shall contain at least the following attributes:

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

- 3168 • The *trace-name* attribute is a character string defined by the trace controller, and that
3169 identifies the trace stream.
- 3170 • The *creation-time* attribute represents the time of the creation of the trace stream.
- 3171 • The *clock-resolution* attribute defines the clock resolution of the clock used to generate
3172 timestamps.
- 3173 • The *stream-min-size* attribute defines the minimum size in bytes of the trace stream strictly
3174 reserved for the trace events.
- 3175 • The *stream-full-policy* attribute defines the policy followed when the trace stream is full; its
3176 value is POSIX_TRACE_LOOP, POSIX_TRACE_UNTIL_FULL, or POSIX_TRACE_FLUSH.
- 3177 • The *max-data-size* attribute defines the maximum record size in bytes of a trace event.

3178 In addition, if the Trace option and the Trace Inherit option are both supported, the
3179 **posix_trace_attr_t** trace stream creation attributes object shall contain at least the following
3180 attributes:

- 3181 • The *inheritance* attribute specifies whether a newly created trace stream will inherit tracing in
3182 its parent's process trace stream. It is either POSIX_TRACE_INHERITED or
3183 POSIX_TRACE_CLOSE_FOR_CHILD.

3184 In addition, if the Trace option and the Trace Log option are both supported, the
3185 **posix_trace_attr_t** trace stream creation attributes object shall contain at least the following
3186 attribute:

- 3187 • If the file type corresponding to the trace log supports the POSIX_TRACE_LOOP or the
3188 POSIX_TRACE_UNTIL_FULL policies, the *log-max-size* attribute defines the maximum size
3189 in bytes of the trace log associated with an active trace stream. Other stream data—for
3190 example, trace attribute values—shall not be included in this size.
- 3191 • The *log-full-policy* attribute defines the policy of a trace log associated with an active trace
3192 stream to be POSIX_TRACE_LOOP, POSIX_TRACE_UNTIL_FULL, or
3193 POSIX_TRACE_APPEND.

3194 2.11.2 Trace Event Type Definitions

3195 2.11.2.1 System Trace Event Type Definitions

3196 The following system trace event types, defined in the <**trace.h**> header, track the invocation of
3197 the trace operations:

- 3198 • POSIX_TRACE_START shall be associated with a trace start operation.
- 3199 • POSIX_TRACE_STOP shall be associated with a trace stop operation.
- 3200 • If the Trace Event Filter option is supported, POSIX_TRACE_FILTER shall be associated with
3201 a trace event type filter change operation.

3202 The following system trace event types, defined in the <**trace.h**> header, report operational trace
3203 events:

- 3204 • POSIX_TRACE_OVERFLOW shall mark the beginning of a trace overflow condition.
- 3205 • POSIX_TRACE_RESUME shall mark the end of a trace overflow condition.
- 3206 • If the Trace Log option is supported, POSIX_TRACE_FLUSH_START shall mark the
3207 beginning of a flush operation.

- 3208 • If the Trace Log option is supported, POSIX_TRACE_FLUSH_STOP shall mark the end of a
 3209 flush operation.
 3210 • If an implementation-defined trace error condition is reported, it shall be marked
 3211 POSIX_TRACE_ERROR.

3212 The interpretation of a trace stream or a trace log by a trace analyzer process relies on the
 3213 information recorded for each trace event, and also on system trace events that indicate the
 3214 invocation of trace control operations and trace system operational trace events.

3215 The POSIX_TRACE_START and POSIX_TRACE_STOP trace events specify the time windows
 3216 during which the trace stream is running.

- 3217 • The POSIX_TRACE_STOP trace event with an associated data that is equal to zero indicates
 3218 a call of the function *posix_trace_stop()*.
 3219 • The POSIX_TRACE_STOP trace event with an associated data that is different from zero
 3220 indicates an automatic stop of the trace stream (see *posix_trace_attr_getstreamfullpolicy()*).

3221 The POSIX_TRACE_FILTER trace event indicates that a trace event type filter value changed
 3222 while the trace stream was running.

3223 The POSIX_TRACE_ERROR serves to inform the analyzer process that an implementation-
 3224 defined internal error of the trace system occurred.

3225 The POSIX_TRACE_OVERFLOW trace event shall be reported with a timestamp equal to the
 3226 timestamp of the first trace event overwritten. This is an indication that some generated trace
 3227 events have been lost.

3228 The POSIX_TRACE_RESUME trace event shall be reported with a timestamp equal to the
 3229 timestamp of the first valid trace event reported after the overflow condition ends and shall be
 3230 reported before this first valid trace event. This is an indication that the trace system is reliably
 3231 recording trace events after an overflow condition.

3232 Each of these trace event types shall be defined by a constant trace event name and a
 3233 **trace_event_id_t** constant; trace event data is associated with some of these trace events.

3234 If the Trace option is supported and the Trace Event Filter option and the Trace Log option are
 3235 not supported, the following predefined system trace events in Table 2-3 shall be defined:

3236 **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.

3246 If the Trace option and the Trace Event Filter option are both supported, and if the Trace Log
 3247 option is not supported, the following predefined system trace events in Table 2-4 (on page 79)
 3248 shall be defined:

3249

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
posix_trace_overflow	POSIX_TRACE_OVERFLOW	trace_event_set_t
		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 80) shall be defined:

3280

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

3281

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.

3282

3283

3284

3285

3286

3287

3288

3289

3290

3291

3292

3293

3294

3295

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:

3303

Table 2-7 Trace Option: User Trace Event

3304

3305

Event Name	Constant
posix_trace_unnamed_userevent	POSIX_TRACE_UNNAMED_USEREVENT

3306

2.11.3 Trace Functions

3307

3308

3309

3310

3311

3312

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_getmaxusereventsizes()*)

- 3319 — Trace stream creation, flush, and shutdown (*posix_trace_create()*)
 3320 — Trace stream and trace log clear (*posix_trace_clear()*)
 3321 — Trace event type identifier manipulation (*posix_trace_trid_eventid_open()*)
 3322 — Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id()*)
 3323 — Trace event type set manipulation (*posix_trace_eventset_empty()*)
 3324 — Trace event type filter set (*posix_trace_set_filter()*)
 3325 — Trace stream start and stop (*posix_trace_start()*)
 3326 — Trace stream information and status read (*posix_trace_get_attr()*)
 3327 • A traced process requires functions to instrument trace points:
 3328 — Trace event type identifiers definition and trace points insertion (*posix_trace_event()*)
 3329 • A trace analyzer process requires functions to retrieve information from a trace stream and
 3330 trace log:
 3331 — Identification information read (*posix_trace_attr_getgenversion()*)
 3332 — Trace system behavior information read (*posix_trace_attr_getinherited()*)
 3333 — Trace stream and trace log size get (*posix_trace_attr_getmaxusereventsiz()*)
 3334 — Trace event type identifier manipulation (*posix_trace_trid_eventid_open()*)
 3335 — Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id()*)
 3336 — Trace log open, rewind, and close (*posix_trace_open()*)
 3337 — Trace stream information and status read (*posix_trace_get_attr()*)
 3338 — Trace event read (*posix_trace_getnext_event()*)

3339 2.12 Data Types

3340 All of the data types used by various functions are defined by the implementation. The
 3341 following table describes some of these types. Other types referenced in the description of a
 3342 function, not mentioned here, can be found in the appropriate header for that function.

3344 Defined Type	3345 Description
3345 cc_t	Type used for terminal special characters.
3346 clock_t	Integer or real-floating type used for processor times, as defined in the ISO C standard.
3347 clockid_t	Used for clock ID type in some timer functions.
3348 dev_t	Arithmetic type used for device numbers.
3349 DIR	Type representing a directory stream.
3350 div_t	Structure type returned by the <i>div()</i> function.
3351 FILE	Structure containing information about a file.
3352 glob_t	Structure type used in pathname pattern matching.
3353 fpos_t	Type containing all information needed to specify uniquely every

Defined Type	Description
3355	position within a file.
3356	3357 gid_t 3358 Integer type used for group IDs. 3359 iconv_t 3360 Type used for conversion descriptors. 3361 id_t 3362 Integer type used as a general identifier; can be used to contain 3363 at least the largest of a pid_t , uid_t , or gid_t . 3364 ino_t 3365 Unsigned integer type used for file serial numbers. 3366 key_t 3367 Arithmetic type used for XSI interprocess communication. 3368 ldiv_t 3369 Structure type returned by the <i>ldiv()</i> function. 3370 mode_t 3371 Integer type used for file attributes. 3372 mqd_t 3373 Used for message queue descriptors. 3374 nfds_t 3375 Integer type used for the number of file descriptors. 3376 nlink_t 3377 Integer type used for link counts. 3378 off_t 3379 Signed integer type used for file sizes. 3380 pid_t 3381 Signed integer type used for process and process group IDs. 3382 pthread_attr_t 3383 Used to identify a thread attribute object. 3384 pthread_cond_t 3385 Used for condition variables. 3386 pthread_condattr_t 3387 Used to identify a condition attribute object. 3388 pthread_key_t 3389 Used for thread-specific data keys. 3390 pthread_mutex_t 3391 Used for mutexes. 3392 pthread_mutexattr_t 3393 Used to identify a mutex attribute object. 3394 pthread_once_t 3395 Used for dynamic package initialization. 3396 pthread_rwlock_t 3397 Used for read-write locks. 3398 pthread_rwlockattr_t 3399 Used for read-write lock attributes. 3400 pthread_t 3401 Used to identify a thread. 3402 ptrdiff_t 3403 Signed integer type of the result of subtracting two pointers. 3404 regex_t 3405 Structure type used in regular expression matching. 3406 regmatch_t 3407 Structure type used in regular expression matching. 3408 rlim_t 3409 Unsigned integer type used for limit values, to which objects of 3410 type int and off_t can be cast without loss of value. 3411 sem_t 3412 Type used in performing semaphore operations. 3413 sig_atomic_t 3414 Integer type of an object that can be accessed as an atomic 3415 entity, even in the presence of asynchronous interrupts. 3416 sigset_t 3417 Integer or structure type of an object used to represent sets 3418 of signals. 3419 size_t 3420 Unsigned integer type used for size of objects. 3421 speed_t 3422 Type used for terminal baud rates. 3423 ssize_t 3424 Signed integer type used for a count of bytes or an error 3425 indication. 3426 suseconds_t 3427 Signed integer type used for time in microseconds. 3428 tcflag_t 3429 Type used for terminal modes. 3430 time_t 3431 Integer or real-floating type used for time in seconds, as defined in 3432 the ISO C standard. 3433 timer_t 3434 Used for timer ID returned by the <i>timer_create()</i> function. 3435 uid_t 3436 Integer type used for user IDs. 3437 useconds_t 3438 Unsigned integer type used for time in microseconds. 3439 va_list 3440 Type used for traversing variable argument lists. 3441 wchar_t 3442 Integer type whose range of values can represent distinct codes for

3404
3405
3406
3407
3408
3409
3410
3411

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

3412

System Interfaces

3414
3415

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

3416 **NAME**
3417 FD_CLR — macros for synchronous I/O multiplexing

3418 **SYNOPSIS**
3419 #include <sys/time.h>
3420 FD_CLR(int fd, fd_set *fdset);
3421 FD_ISSET(int fd, fd_set *fdset);
3422 FD_SET(int fd, fd_set *fdset);
3423 FD_ZERO(fd_set *fdset);

3424 **DESCRIPTION**
3425 Refer to *pselect()*.

3426 **NAME**
3427 _Exit, _exit — terminate a process

3428 **SYNOPSIS**
3429 #include <stdlib.h>
3430 void _Exit(int *status*);
3431 #include <unistd.h>
3432 void _exit(int *status*);

3433 **DESCRIPTION**
3434 Refer to *exit()*.

3435 NAME

3436 _longjmp, _setjmp — non-local goto

3437 SYNOPSIS

3438 XSI #include <setjmp.h>

3439 void _longjmp(jmp_buf env, int val);

3440 int _setjmp(jmp_buf env);

3441

3442 DESCRIPTION

3443 The *_longjmp()* and *_setjmp()* functions shall be equivalent to *longjmp()* and *setjmp()*, respectively, with the additional restriction that *_longjmp()* and *_setjmp()* shall not manipulate the signal mask.

3446 If *_longjmp()* is called even though *env* was never initialized by a call to *_setjmp()*, or when the
3447 last such call was in a function that has since returned, the results are undefined.

3448 RETURN VALUE

3449 Refer to *longjmp()* and *setjmp()*.

3450 ERRORS

3451 No errors are defined.

3452 EXAMPLES

3453 None.

3454 APPLICATION USAGE

3455 If *_longjmp()* is executed and the environment in which *_setjmp()* was executed no longer exists,
3456 errors can occur. The conditions under which the environment of the *_setjmp()* no longer exists
3457 include exiting the function that contains the *_setjmp()* call, and exiting an inner block with
3458 temporary storage. This condition might not be detectable, in which case the *_longjmp()* occurs
3459 and, if the environment no longer exists, the contents of the temporary storage of an inner block
3460 are unpredictable. This condition might also cause unexpected process termination. If the
3461 function has returned, the results are undefined.

3462 Passing *longjmp()* a pointer to a buffer not created by *setjmp()*, passing *_longjmp()* a pointer to a buffer
3463 not created by *_setjmp()*, passing *siglongjmp()* a pointer to a buffer not created by
3464 *sigsetjmp()*, or passing any of these three functions a buffer that has been modified by the user
3465 can cause all the problems listed above, and more.

3466 The *_longjmp()* and *_setjmp()* functions are included to support programs written to historical
3467 system interfaces. New applications should use *siglongjmp()* and *sigsetjmp()* respectively.

3468 RATIONALE

3469 None.

3470 FUTURE DIRECTIONS

3471 The *_longjmp()* and *_setjmp()* functions may be marked LEGACY in a future version.

3472 SEE ALSO

3473 *longjmp()*, *setjmp()*, *siglongjmp()*, *sigsetjmp()*, the Base Definitions volume of
3474 IEEE Std 1003.1-2001, <setjmp.h>

3475 CHANGE HISTORY

3476 First released in Issue 4, Version 2.

3477 **Issue 5**

3478 Moved from X/OPEN UNIX extension to BASE.

3479 NAME

3480 *_tolower* — transliterate uppercase characters to lowercase

3481 SYNOPSIS

3482 XSI #include <ctype.h>

3483 int _tolower(int *c*);

3484

3485 DESCRIPTION

3486 The *_tolower()* macro shall be equivalent to *tolower(c)* except that the application shall ensure
3487 that the argument *c* is an uppercase letter.

3488 RETURN VALUE

3489 Upon successful completion, *_tolower()* shall return the lowercase letter corresponding to the
3490 argument passed.

3491 ERRORS

3492 No errors are defined.

3493 EXAMPLES

3494 None.

3495 APPLICATION USAGE

3496 None.

3497 RATIONALE

3498 None.

3499 FUTURE DIRECTIONS

3500 None.

3501 SEE ALSO

3502 *tolower()*, *isupper()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
3503 <ctype.h>

3504 CHANGE HISTORY

3505 First released in Issue 1. Derived from Issue 1 of the SVID.

3506 Issue 6

3507 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

3508 NAME

3509 _toupper — transliterate lowercase characters to uppercase

3510 SYNOPSIS

3511 XSI #include <ctype.h>

3512 int _toupper(int c);

3513

3514 DESCRIPTION

3515 The *_toupper()* macro shall be equivalent to *toupper()* except that the application shall ensure
3516 that the argument *c* is a lowercase letter.

3517 RETURN VALUE

3518 Upon successful completion, *_toupper()* shall return the uppercase letter corresponding to the
3519 argument passed.

3520 ERRORS

3521 No errors are defined.

3522 EXAMPLES

3523 None.

3524 APPLICATION USAGE

3525 None.

3526 RATIONALE

3527 None.

3528 FUTURE DIRECTIONS

3529 None.

3530 SEE ALSO

3531 *islower()*, *toupper()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
3532 <ctype.h>

3533 CHANGE HISTORY

3534 First released in Issue 1. Derived from Issue 1 of the SVID.

3535 Issue 6

3536 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

3537 **NAME**

3538 a64l, l64a — convert between a 32-bit integer and a radix-64 ASCII string

3539 **SYNOPSIS**

3540 XSI #include <stdlib.h>

```
3541      long a64l(const char *s);  
3542      char *l64a(long value);
```

3543

3544 **DESCRIPTION**

3545 These functions maintain numbers stored in radix-64 ASCII characters. This is a notation by
3546 which 32-bit integers can be represented by up to six characters; each character represents a digit
3547 in radix-64 notation. If the type **long** contains more than 32 bits, only the low-order 32 bits shall
3548 be used for these operations.

3549 The characters used to represent digits are '.' (dot) for 0, '/' for 1, '0' through '9' for [2,11],
3550 'A' through 'Z' for [12,37], and 'a' through 'z' for [38,63].

3551 The *a64l()* function shall take a pointer to a radix-64 representation, in which the first digit is the
3552 least significant, and return the corresponding **long** value. If the string pointed to by *s* contains
3553 more than six characters, *a64l()* shall use the first six. If the first six characters of the string
3554 contain a null terminator, *a64l()* shall use only characters preceding the null terminator. The
3555 *a64l()* function shall scan the character string from left to right with the least significant digit on
3556 the left, decoding each character as a 6-bit radix-64 number. If the type **long** contains more than
3557 32 bits, the resulting value is sign-extended. The behavior of *a64l()* is unspecified if *s* is a null
3558 pointer or the string pointed to by *s* was not generated by a previous call to *l64a()*.

3559 The *l64a()* function shall take a **long** argument and return a pointer to the corresponding radix-
3560 64 representation. The behavior of *l64a()* is unspecified if *value* is negative.

3561 The value returned by *l64a()* may be a pointer into a static buffer. Subsequent calls to *l64a()* may
3562 overwrite the buffer.

3563 The *l64a()* function need not be reentrant. A function that is not required to be reentrant is not
3564 required to be thread-safe.

3565 **RETURN VALUE**

3566 Upon successful completion, *a64l()* shall return the **long** value resulting from conversion of the
3567 input string. If a string pointed to by *s* is an empty string, *a64l()* shall return 0L.

3568 The *l64a()* function shall return a pointer to the radix-64 representation. If *value* is 0L, *l64a()* shall
3569 return a pointer to an empty string.

3570 **ERRORS**

3571 No errors are defined.

3572 **EXAMPLES**

3573 None.

3574 **APPLICATION USAGE**

3575 If the type **long** contains more than 32 bits, the result of *a64l/l64a(x)* is *x* in the low-order 32 bits.

3576 **RATIONALE**

3577 This is not the same encoding as used by either encoding variant of the *uuencode* utility.

3578 **FUTURE DIRECTIONS**

3579 None.

3580 **SEE ALSO**3581 *strtoul()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**stdlib.h**>, the Shell and Utilities
3582 volume of IEEE Std 1003.1-2001, *uuencode*3583 **CHANGE HISTORY**

3584 First released in Issue 4, Version 2.

3585 **Issue 5**

3586 Moved from X/OPEN UNIX extension to BASE.

3587 Normative text previously in the APPLICATION USAGE section is moved to the
3588 DESCRIPTION.

3589 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

3590 **NAME**

3591 abort — generate an abnormal process abort

3592 **SYNOPSIS**

```
3593     #include <stdlib.h>
3594
3595     void abort(void);
```

3595 **DESCRIPTION**

3596 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 3597 conflict between the requirements described here and the ISO C standard is unintentional. This
 3598 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

3599 The *abort()* function shall cause abnormal process termination to occur, unless the signal
 3600 SIGABRT is being caught and the signal handler does not return.

3601 CX The abnormal termination processing shall include the default actions defined for SIGABRT and 1
 3602 may include an attempt to effect *fclose()* on all open streams. 1

3603 The SIGABRT signal shall be sent to the calling process as if by means of *raise()* with the
 3604 argument SIGABRT.

3605 CX The status made available to *wait()* or *waitpid()* by *abort()* shall be that of a process terminated
 3606 by the SIGABRT signal. The *abort()* function shall override blocking or ignoring the SIGABRT
 3607 signal.

3608 **RETURN VALUE**3609 The *abort()* function shall not return.3610 **ERRORS**

3611 No errors are defined.

3612 **EXAMPLES**

3613 None.

3614 **APPLICATION USAGE**

3615 Catching the signal is intended to provide the application writer with a portable means to abort
 3616 processing, free from possible interference from any implementation-supplied functions. 2

3617 **RATIONALE**

3618 The ISO/IEC 9899:1999 standard requires the *abort()* function to be async-signal-safe. Since 1
 3619 IEEE Std 1003.1-2001 defers to the ISO C standard, this required a change to the DESCRIPTION 1
 3620 from “shall include the effect of *fclose()*” to “may include an attempt to effect *fclose()*.”. 1

3621 The revised wording permits some backwards-compatibility and avoids a potential deadlock 1
 3622 situation. 1

3623 The Open Group Base Resolution bwg2002-003 is applied, removing the following XSI shaded 1
 3624 paragraph from the DESCRIPTION: 1

3625 “On XSI-conformant systems, in addition the abnormal termination processing shall include the 1
 3626 effect of *fclose()* on message catalog descriptors.” 1

3627 There were several reasons to remove this paragraph: 1

- 3628 • No special processing of open message catalogs needs to be performed prior to abnormal 1
 3629 process termination. 1

- 3630 • The main reason to specifically mention that *abort()* includes the effect of *fclose()* on open 1
 3631 streams is to flush output queued on the stream. Message catalogs in this context are read- 1
 3632 only and, therefore, do not need to be flushed. 1

- 3633 • The effect of *fclose()* on a message catalog descriptor is unspecified. Message catalog 1
3634 descriptors are allowed, but not required to be implemented using a file descriptor, but there 1
3635 is no mention in IEEE Std 1003.1-2001 of a message catalog descriptor using a standard I/O 1
3636 stream FILE object as would be expected by *fclose()*. 1

3637 **FUTURE DIRECTIONS**

3638 None.

3639 **SEE ALSO**

3640 *exit()*, *kill()*, *raise()*, *signal()*, *wait()*, *waitpid()*, the Base Definitions volume of 1
3641 IEEE Std 1003.1-2001, <stdlib.h> 1

3642 **CHANGE HISTORY**

3643 First released in Issue 1. Derived from Issue 1 of the SVID.

3644 **Issue 6**

3645 Extensions beyond the ISO C standard are marked.

3646 Changes are made to the DESCRIPTION for alignment with the ISO/IEC 9899:1999 standard.

3647 The Open Group Base Resolution bwg2002-003 is applied.

3648 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/10 is applied, changing the 1
3649 DESCRIPTION of abnormal termination processing and adding to the RATIONALE section. 1

3650 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/9 is applied, changing “implementation- 2
3651 defined functions” to “implementation-supplied functions” in the APPLICATION USAGE 2
3652 section. 2

3653 NAME

3654 **abs** — return an integer absolute value

3655 SYNOPSIS

```
3656       #include <stdlib.h>
3657
3658       int abs(int i);
```

3658 DESCRIPTION

3659 CX The functionality described on this reference page is aligned with the ISO C standard. Any
3660 conflict between the requirements described here and the ISO C standard is unintentional. This
3661 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

3662 The *abs()* function shall compute the absolute value of its integer operand, *i*. If the result cannot
3663 be represented, the behavior is undefined.

3664 RETURN VALUE

3665 The *abs()* function shall return the absolute value of its integer operand.

3666 ERRORS

3667 No errors are defined.

3668 EXAMPLES

3669 None.

3670 APPLICATION USAGE

3671 In two's-complement representation, the absolute value of the negative integer with largest
3672 magnitude {INT_MIN} might not be representable.

3673 RATIONALE

3674 None.

3675 FUTURE DIRECTIONS

3676 None.

3677 SEE ALSO

3678 *fabs()*, *labs()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**stdlib.h**>

3679 CHANGE HISTORY

3680 First released in Issue 1. Derived from Issue 1 of the SVID.

3681 Issue 6

3682 Extensions beyond the ISO C standard are marked.

3683 NAME

3684 accept — accept a new connection on a socket

3685 SYNOPSIS

```
3686 #include <sys/socket.h>
3687 int accept(int socket, struct sockaddr *restrict address,
3688             socklen_t *restrict address_len);
```

3689 DESCRIPTION

3690 The *accept()* function shall extract the first connection on the queue of pending connections,
3691 create a new socket with the same socket type protocol and address family as the specified
3692 socket, and allocate a new file descriptor for that socket.

3693 The *accept()* function takes the following arguments:

3694 *socket* Specifies a socket that was created with *socket()*, has been bound to an address
3695 with *bind()*, and has issued a successful call to *listen()*.

3696 *address* Either a null pointer, or a pointer to a **sockaddr** structure where the address of
3697 the connecting socket shall be returned.

3698 *address_len* Points to a **socklen_t** structure which on input specifies the length of the
3699 supplied **sockaddr** structure, and on output specifies the length of the stored
3700 address.

3701 If *address* is not a null pointer, the address of the peer for the accepted connection shall be stored
3702 in the **sockaddr** structure pointed to by *address*, and the length of this address shall be stored in
3703 the object pointed to by *address_len*.

3704 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
3705 the stored address shall be truncated.

3706 If the protocol permits connections by unbound clients, and the peer is not bound, then the value
3707 stored in the object pointed to by *address* is unspecified.

3708 If the listen queue is empty of connection requests and O_NONBLOCK is not set on the file
3709 descriptor for the socket, *accept()* shall block until a connection is present. If the *listen()* queue is
3710 empty of connection requests and O_NONBLOCK is set on the file descriptor for the socket,
3711 *accept()* shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

3712 The accepted socket cannot itself accept more connections. The original socket remains open and
3713 can accept more connections.

3714 RETURN VALUE

3715 Upon successful completion, *accept()* shall return the non-negative file descriptor of the accepted
3716 socket. Otherwise, -1 shall be returned and *errno* set to indicate the error.

3717 ERRORS

3718 The *accept()* function shall fail if:

3719 [EAGAIN] or [EWOULDBLOCK]
3720 O_NONBLOCK is set for the socket file descriptor and no connections are
3721 present to be accepted.

3722 [EBADF] The *socket* argument is not a valid file descriptor.

3723 [ECONNABORTED]
3724 A connection has been aborted.

3725	[EINTR]	The <i>accept()</i> function was interrupted by a signal that was caught before a valid connection arrived.
3726		
3727	[EINVAL]	The <i>socket</i> is not accepting connections.
3728	[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
3729	[ENFILE]	The maximum number of file descriptors in the system are already open.
3730	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
3731	[EOPNOTSUPP]	The socket type of the specified socket does not support accepting connections.
3732		
3733		The <i>accept()</i> function may fail if:
3734	[ENOBUFS]	No buffer space is available.
3735	[ENOMEM]	There was insufficient memory available to complete the operation.
3736 XSR	[EPROTO]	A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized.
3737		

3738 EXAMPLES

3739 None.

3740 APPLICATION USAGE

3741 When a connection is available, *select()* indicates that the file descriptor for the socket is ready
3742 for reading.

3743 RATIONALE

3744 None.

3745 FUTURE DIRECTIONS

3746 None.

3747 SEE ALSO

3748 *bind()*, *connect()*, *listen()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001,
3749 <sys/socket.h>

3750 CHANGE HISTORY

3751 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

3752 The **restrict** keyword is added to the *accept()* prototype for alignment with the
3753 ISO/IEC 9899:1999 standard.

3754 NAME

3755 access — determine accessibility of a file

3756 SYNOPSIS

```
3757 #include <unistd.h>
3758 int access(const char *path, int amode);
```

3759 DESCRIPTION

3760 The *access()* function shall check the file named by the *pathname* pointed to by the *path* argument for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID.

3763 The value of *amode* is either the bitwise-inclusive OR of the access permissions to be checked (R_OK, W_OK, X_OK) or the existence test (F_OK).

3765 If any access permissions are checked, each shall be checked individually, as described in the
3766 Base Definitions volume of IEEE Std 1003.1-2001, Chapter 3, Definitions. If the process has
3767 appropriate privileges, an implementation may indicate success for X_OK even if none of the
3768 execute file permission bits are set.

3769 RETURN VALUE

3770 If the requested access is permitted, *access()* succeeds and shall return 0; otherwise, -1 shall be
3771 returned and *errno* shall be set to indicate the error.

3772 ERRORS

3773 The *access()* function shall fail if:

3774 [EACCES] Permission bits of the file mode do not permit the requested access, or search
3775 permission is denied on a component of the path prefix.

3776 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
3777 argument.

3778 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
3779 component is longer than {NAME_MAX}.

3781 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

3782 [ENOTDIR] A component of the path prefix is not a directory.

3783 [EROFS] Write access is requested for a file on a read-only file system.

3784 The *access()* function may fail if:

3785 [EINVAL] The value of the *amode* argument is invalid.

3786 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
3787 resolution of the *path* argument.

3788 [ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the *path* argument,
3789 the length of the substituted pathname string exceeded {PATH_MAX}.

3791 [ETXTBSY] Write access is requested for a pure procedure (shared text) file that is being
3792 executed.

3793 EXAMPLES

3794 Testing for the Existence of a File

3795 The following example tests whether a file named **myfile** exists in the **/tmp** directory.

```
3796 #include <unistd.h>
3797 ...
3798 int result;
3799 const char *filename = "/tmp/myfile";
3800 result = access (filename, F_OK);
```

3801 APPLICATION USAGE

3802 Additional values of *amode* other than the set defined in the description may be valid; for
3803 example, if a system has extended access controls.

3804 RATIONALE

3805 In early proposals, some inadequacies in the *access()* function led to the creation of an *eaccess()*
3806 function because:

- 3807 1. Historical implementations of *access()* do not test file access correctly when the process'
3808 real user ID is superuser. In particular, they always return zero when testing execute
3809 permissions without regard to whether the file is executable.
- 3810 2. The superuser has complete access to all files on a system. As a consequence, programs
3811 started by the superuser and switched to the effective user ID with lesser privileges cannot
3812 use *access()* to test their file access permissions.

3813 However, the historical model of *eaccess()* does not resolve problem (1), so this volume of
3814 IEEE Std 1003.1-2001 now allows *access()* to behave in the desired way because several
3815 implementations have corrected the problem. It was also argued that problem (2) is more easily
3816 solved by using *open()*, *chdir()*, or one of the *exec* functions as appropriate and responding to the
3817 error, rather than creating a new function that would not be as reliable. Therefore, *eaccess()* is not
3818 included in this volume of IEEE Std 1003.1-2001.

3819 The sentence concerning appropriate privileges and execute permission bits reflects the two
3820 possibilities implemented by historical implementations when checking superuser access for
3821 **X_OK**.

3822 New implementations are discouraged from returning **X_OK** unless at least one execution
3823 permission bit is set.

3824 FUTURE DIRECTIONS

3825 None.

3826 SEE ALSO

3827 *chmod()*, *stat()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

3828 CHANGE HISTORY

3829 First released in Issue 1. Derived from Issue 1 of the SVID.

3830 Issue 6

3831 The following new requirements on POSIX implementations derive from alignment with the
3832 Single UNIX Specification:

- 3833 • The [ELOOP] mandatory error condition is added.
- 3834 • A second [ENAMETOOLONG] is added as an optional error condition.

3835 • The [ETXTBSY] optional error condition is added.

3836 The following changes were made to align with the IEEE P1003.1a draft standard:

3837 • The [ELOOP] optional error condition is added.

3838 **NAME**

3839 acos, acosf, acosl — arc cosine functions

3840 **SYNOPSIS**

```
3841     #include <math.h>
3842
3843     double acos(double x);
3844     float acosf(float x);
3845     long double acosl(long double x);
```

3845 **DESCRIPTION**

3846 CX The functionality described on this reference page is aligned with the ISO C standard. Any
3847 conflict between the requirements described here and the ISO C standard is unintentional. This
3848 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

3849 These functions shall compute the principal value of the arc cosine of their argument x . The
3850 value of x should be in the range $[-1,1]$.

3851 An application wishing to check for error situations should set $errno$ to zero and call
3852 $feclearexcept(FE_ALL_EXCEPT)$ before calling these functions. On return, if $errno$ is non-zero or
3853 $fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)$ is non-
3854 zero, an error has occurred.

3855 **RETURN VALUE**

3856 Upon successful completion, these functions shall return the arc cosine of x , in the range $[0,\pi]$
3857 radians.

3858 MX For finite values of x not in the range $[-1,1]$, a domain error shall occur, and either a NaN (if
3859 supported), or an implementation-defined value shall be returned.

3860 MX If x is NaN, a NaN shall be returned.

3861 If x is +1, +0 shall be returned.

3862 If x is $\pm\infty$, a domain error shall occur, and either a NaN (if supported), or an implementation-
3863 defined value shall be returned.

3864 **ERRORS**

3865 These functions shall fail if:

3866 MX Domain Error The x argument is finite and is not in the range $[-1,1]$, or is $\pm\infty$.

3867 If the integer expression $(math_errhandling \& MATH_ERRNO)$ is non-zero,
3868 then $errno$ shall be set to [EDOM]. If the integer expression $(math_errhandling \&$
3869 $MATH_ERREXCEPT)$ is non-zero, then the invalid floating-point exception
3870 shall be raised.

3871 **EXAMPLES**

3872 None.

3873 **APPLICATION USAGE**

3874 On error, the expressions $(math_errhandling \& MATH_ERRNO)$ and $(math_errhandling \&$
3875 $MATH_ERREXCEPT)$ are independent of each other, but at least one of them must be non-zero.

3876 **RATIONALE**

3877 None.

3878 **FUTURE DIRECTIONS**

3879 None.

3880 **SEE ALSO**3881 *cos()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
3882 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>3883 **CHANGE HISTORY**

3884 First released in Issue 1. Derived from Issue 1 of the SVID.

3885 **Issue 5**3886 The DESCRIPTION is updated to indicate how an application should check for an error. This
3887 text was previously published in the APPLICATION USAGE section.3888 **Issue 6**3889 The *acosf()* and *acosl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.3890 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
3891 revised to align with the ISO/IEC 9899:1999 standard.3892 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
3893 marked.

3894 **NAME**

3895 acosh, acoshf, acoshl — inverse hyperbolic cosine functions

3896 **SYNOPSIS**

```
3897 #include <math.h>
3898 double acosh(double x);
3899 float acoshf(float x);
3900 long double acoshl(long double x);
```

3901 **DESCRIPTION**

3902 CX The functionality described on this reference page is aligned with the ISO C standard. Any
3903 conflict between the requirements described here and the ISO C standard is unintentional. This
3904 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

3905 These functions shall compute the inverse hyperbolic cosine of their argument *x*.

3906 An application wishing to check for error situations should set *errno* to zero and call
3907 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
3908 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
3909 zero, an error has occurred.

3910 **RETURN VALUE**

3911 Upon successful completion, these functions shall return the inverse hyperbolic cosine of their
3912 argument.

3913 MX For finite values of *x* < 1, a domain error shall occur, and either a NaN (if supported), or an
3914 implementation-defined value shall be returned.

3915 MX If *x* is NaN, a NaN shall be returned.

3916 If *x* is +1, +0 shall be returned.

3917 If *x* is +Inf, +Inf shall be returned.

3918 If *x* is -Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-
3919 defined value shall be returned.

3920 **ERRORS**

3921 These functions shall fail if:

3922 MX Domain Error The *x* argument is finite and less than +1.0, or is -Inf.

3923 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
3924 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
3925 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
3926 shall be raised.

3927 **EXAMPLES**

3928 None.

3929 **APPLICATION USAGE**

3930 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
3931 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

3932 **RATIONALE**

3933 None.

3934 **FUTURE DIRECTIONS**

3935 None.

3936 **SEE ALSO**3937 *cosh()*, *feclearexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section
3938 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>3939 **CHANGE HISTORY**

3940 First released in Issue 4, Version 2.

3941 **Issue 5**

3942 Moved from X/OPEN UNIX extension to BASE.

3943 **Issue 6**3944 The *acosh()* function is no longer marked as an extension.3945 The *acoshf()* and *acoshl()* functions are added for alignment with the ISO/IEC 9899:1999 1
3946 standard.3947 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
3948 revised to align with the ISO/IEC 9899:1999 standard.3949 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
3950 marked.

```
3951 NAME
3952         acosl — arc cosine functions
3953 SYNOPSIS
3954         #include <math.h>
3955         long double acosl(long double x);
3956 DESCRIPTION
3957         Refer to acos().
```

3958 **NAME**

3959 — cancel an asynchronous I/O request (**REALTIME**)

3960 **SYNOPSIS**

3961 AIO

```
#include <aio.h>
```

3962

```
int aio_cancel(int fildes, struct aiocb *aiocbp);
```

3963

3964 **DESCRIPTION**

3965 The *aio_cancel()* function shall attempt to cancel one or more asynchronous I/O requests
3966 currently outstanding against file descriptor *fildes*. The *aiocbp* argument points to the
3967 asynchronous I/O control block for a particular request to be canceled. If *aiocbp* is NULL, then
3968 all outstanding cancelable asynchronous I/O requests against *fildes* shall be canceled.

3969 Normal asynchronous notification shall occur for asynchronous I/O operations that are
3970 successfully canceled. If there are requests that cannot be canceled, then the normal
3971 asynchronous completion process shall take place for those requests when they are completed.

3972 For requested operations that are successfully canceled, the associated error status shall be set to
3973 [ECANCELED] and the return status shall be -1. For requested operations that are not
3974 successfully canceled, the *aiocbp* shall not be modified by *aio_cancel()*.

3975 If *aiocbp* is not NULL, then if *fildes* does not have the same value as the file descriptor with which
3976 the asynchronous operation was initiated, unspecified results occur.

3977 Which operations are cancelable is implementation-defined.

3978 **RETURN VALUE**

3979 The *aio_cancel()* function shall return the value AIO_CANCELED if the requested operation(s) 2
3980 were canceled. The value AIO_NOTCANCELED shall be returned if at least one of the
3981 requested operation(s) cannot be canceled because it is in progress. In this case, the state of the
3982 other operations, if any, referenced in the call to *aio_cancel()* is not indicated by the return value
3983 of *aio_cancel()*. The application may determine the state of affairs for these operations by using
3984 *aio_error()*. The value AIO_ALLDONE is returned if all of the operations have already
3985 completed. Otherwise, the function shall return -1 and set *errno* to indicate the error.

3986 **ERRORS**

3987 The *aio_cancel()* function shall fail if:

3988 [EBADF] The *fildes* argument is not a valid file descriptor.

3989 **EXAMPLES**

3990 None.

3991 **APPLICATION USAGE**

3992 The *aio_cancel()* function is part of the Asynchronous Input and Output option and need not be
3993 available on all implementations.

3994 **RATIONALE**

3995 None.

3996 **FUTURE DIRECTIONS**

3997 None.

3998 **SEE ALSO**

3999 , , the Base Definitions volume of IEEE Std 1003.1-2001, <aio.h>

4000 **CHANGE HISTORY**

4001 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4002 **Issue 6**4003 The [ENOSYS] error condition has been removed as stubs need not be provided if an
4004 implementation does not support the Asynchronous Input and Output option.

4005 The APPLICATION USAGE section is added.

4006 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/10 is applied, removing the words “to the 2
4007 calling process” in the RETURN VALUE section. The term was unnecessary and precluded 2
4008 threads. 2

4009 NAME

4010 aio_error — retrieve errors status for an asynchronous I/O operation (**REALTIME**)

4011 SYNOPSIS

4012 AIO #include <aio.h>

4013 int aio_error(const struct aiocb *aiocbp);

4014

4015 DESCRIPTION

4016 The *aio_error()* function shall return the error status associated with the **aiocb** structure
4017 referenced by the *aiocbp* argument. The error status for an asynchronous I/O operation is the
4018 SIO *errno* value that would be set by the corresponding *read()*, *write()*, *fdatasync()*, or *fsync()*
4019 operation. If the operation has not yet completed, then the error status shall be equal to
4020 [EINPROGRESS].

4021 RETURN VALUE

4022 If the asynchronous I/O operation has completed successfully, then 0 shall be returned. If the
4023 asynchronous operation has completed unsuccessfully, then the error status, as described for
4024 SIO *read()*, *write()*, *fdatasync()*, and *fsync()*, shall be returned. If the asynchronous I/O operation has
4025 not yet completed, then [EINPROGRESS] shall be returned.

4026 ERRORS

4027 The *aio_error()* function may fail if:

4028 [EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose
4029 return status has not yet been retrieved.

4030 EXAMPLES

4031 None.

4032 APPLICATION USAGE

4033 The *aio_error()* function is part of the Asynchronous Input and Output option and need not be
4034 available on all implementations.

4035 RATIONALE

4036 None.

4037 FUTURE DIRECTIONS

4038 None.

4039 SEE ALSO

4040 *aio_cancel()*, *aio_fsync()*, *aio_read()*, *aio_return()*, *aio_write()*, *close()*, *exec*, *exit()*, *fork()*, *lio_listio()*,
4041 *lseek()*, *read()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**aio.h**>

4042 CHANGE HISTORY

4043 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4044 Issue 6

4045 The [ENOSYS] error condition has been removed as stubs need not be provided if an
4046 implementation does not support the Asynchronous Input and Output option.

4047 The APPLICATION USAGE section is added.

4048 NAME

4049 aio_fsync — asynchronous file synchronization (**REALTIME**)

4050 SYNOPSIS

4051 AIO #include <aio.h>

4052 int aio_fsync(int op, struct aiocb *aiocbp);

4053

4054 DESCRIPTION

4055 The *aio_fsync()* function shall asynchronously force all I/O operations associated with the file
4056 indicated by the file descriptor *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp*
4057 argument and queued at the time of the call to *aio_fsync()* to the synchronized I/O completion
4058 state. The function call shall return when the synchronization request has been initiated or
4059 queued to the file or device (even when the data cannot be synchronized immediately).

4060 If *op* is O_DSYNC, all currently queued I/O operations shall be completed as if by a call to
4061 *fdatasync()*; that is, as defined for synchronized I/O data integrity completion. If *op* is O_SYNC,
4062 all currently queued I/O operations shall be completed as if by a call to *fsync()*; that is, as
4063 defined for synchronized I/O file integrity completion. If the *aio_fsync()* function fails, or if the
4064 operation queued by *aio_fsync()* fails, then, as for *fsync()* and *fdatasync()*, outstanding I/O
4065 operations are not guaranteed to have been completed.

4066 If *aio_fsync()* succeeds, then it is only the I/O that was queued at the time of the call to
4067 *aio_fsync()* that is guaranteed to be forced to the relevant completion state. The completion of
4068 subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized
4069 fashion.

4070 The *aiocbp* argument refers to an asynchronous I/O control block. The *aiocbp* value may be used
4071 as an argument to *aio_error()* and *aio_return()* in order to determine the error status and return
4072 status, respectively, of the asynchronous operation while it is proceeding. When the request is
4073 queued, the error status for the operation is [EINPROGRESS]. When all data has been
4074 successfully transferred, the error status shall be reset to reflect the success or failure of the
4075 operation. If the operation does not complete successfully, the error status for the operation shall
4076 be set to indicate the error. The *aio_sigevent* member determines the asynchronous notification to
4077 occur as specified in Section 2.4.1 (on page 28) when all operations have achieved synchronized
4078 I/O completion. All other members of the structure referenced by *aiocbp* are ignored. If the
4079 control block referenced by *aiocbp* becomes an illegal address prior to asynchronous I/O
4080 completion, then the behavior is undefined.

4081 If the *aio_fsync()* function fails or *aiocbp* indicates an error condition, data is not guaranteed to
4082 have been successfully transferred.

4083 RETURN VALUE

4084 The *aio_fsync()* function shall return the value 0 if the I/O operation is successfully queued; 2
4085 otherwise, the function shall return the value -1 and set *errno* to indicate the error.

4086 ERRORS

4087 The *aio_fsync()* function shall fail if:

- | | |
|------------------------------|--|
| 4088 [EAGAIN] | The requested asynchronous operation was not queued due to temporary
4089 resource limitations. |
| 4090 [EBADF] | The <i>aio_fildes</i> member of the aiocb structure referenced by the <i>aiocbp</i> argument
4091 is not a valid file descriptor open for writing. |
| 4092 [EINVAL] | This implementation does not support synchronized I/O for this file. |

4093 [EINVAL] A value of *op* other than O_DSYNC or O_SYNC was specified.

4094 In the event that any of the queued I/O operations fail, *aio_fsync()* shall return the error
4095 condition defined for *read()* and *write()*. The error is returned in the error status for the
4096 asynchronous *fsync()* operation, which can be retrieved using *aio_error()*.

4097 **EXAMPLES**

4098 None.

4099 **APPLICATION USAGE**

4100 The *aio_fsync()* function is part of the Asynchronous Input and Output option and need not be
4101 available on all implementations.

4102 **RATIONALE**

4103 None.

4104 **FUTURE DIRECTIONS**

4105 None.

4106 **SEE ALSO**

4107 *fcntl()*, *fdatasync()*, *fsync()*, *open()*, *read()*, *write()*, the Base Definitions volume of
4108 IEEE Std 1003.1-2001, <**aio.h**>

4109 **CHANGE HISTORY**

4110 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

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 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/11 is applied, removing the words “to the 2
4116 calling process” in the RETURN VALUE section. The term was unnecessary and precluded 2
4117 threads. 2

4118 NAME

4119 aio_read — asynchronous read from a file (REALTIME)

4120 SYNOPSIS

```
4121 AIO #include <aio.h>
4122     int aio_read(struct aiocb *aiocbp);
```

4124 DESCRIPTION

4125 The *aio_read()* function shall read *aiocbp->aio_nbytes* from the file associated with
 4126 *aiocbp->aio_fildes* into the buffer pointed to by *aiocbp->aio_buf*. The function call shall return when
 4127 the read request has been initiated or queued to the file or device (even when the data cannot be
 4128 delivered immediately).

4129 PIO	If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted	2
4130	at a priority equal to a base scheduling priority minus <i>aiocbp->aio_repprio</i> . If Thread Execution	2
4131	Scheduling is not supported, then the base scheduling priority is that of the calling process;	2
4132 PIO TPS	otherwise, the base scheduling priority is that of the calling thread.	2

4133 The *aiocbp* value may be used as an argument to *aio_error()* and *aio_return()* in order to
 4134 determine the error status and return status, respectively, of the asynchronous operation while it
 4135 is proceeding. If an error condition is encountered during queuing, the function call shall return
 4136 without having initiated or queued the request. The requested operation takes place at the
 4137 absolute position in the file as given by *aio_offset*, as if *Iseek()* were called immediately prior to
 4138 the operation with an *offset* equal to *aio_offset* and a *whence* equal to SEEK_SET. After a
 4139 successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file
 4140 is unspecified.

4141 The *aiocbp->aio_lio_opcode* field shall be ignored by *aio_read()*.

4142 The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp->aio_buf* or
 4143 the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O
 4144 completion, then the behavior is undefined.

4145 Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

4146 SIO If synchronized I/O is enabled on the file associated with *aiocbp->aio_fildes*, the behavior of this
 4147 function shall be according to the definitions of synchronized I/O data integrity completion and
 4148 synchronized I/O file integrity completion.

4149 For any system action that changes the process memory space while an asynchronous I/O is
 4150 outstanding to the address range being changed, the result of that action is undefined.

4151 For regular files, no data transfer shall occur past the offset maximum established in the open
 4152 file description associated with *aiocbp->aio_fildes*.

4153 RETURN VALUE

4154 The *aio_read()* function shall return the value zero if the I/O operation is successfully queued; 2
 4155 otherwise, the function shall return the value -1 and set *errno* to indicate the error.

4156 ERRORS

4157 The *aio_read()* function shall fail if:

4158 [EAGAIN] The requested asynchronous I/O operation was not queued due to system
 4159 resource limitations.

4160 Each of the following conditions may be detected synchronously at the time of the call to
 4161 *aio_read()*, or asynchronously. If any of the conditions below are detected synchronously, the
 4162 *aio_read()* function shall return -1 and set *errno* to the corresponding value. If any of the

4163 conditions below are detected asynchronously, the return status of the asynchronous operation
4164 is set to -1, and the error status of the asynchronous operation is set to the corresponding value.

4165 [EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for reading.

4166 [EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid,
4167 PIO *aiocbp->aio_reqprio* is not a valid value, or *aiocbp->aio_nbytes* is an invalid
4168 value.

4169 In the case that the *aio_read()* successfully queues the I/O operation but the operation is
4170 subsequently canceled or encounters an error, the return status of the asynchronous operation is
4171 one of the values normally returned by the *read()* function call. In addition, the error status of
4172 the asynchronous operation is set to one of the error statuses normally set by the *read()* function
4173 call, or one of the following values:

4174 [EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for reading.

4175 [ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit
4176 *aio_cancel()* request.

4177 [EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid.

4178 The following condition may be detected synchronously or asynchronously:

4179 [EOVERFLOW] The file is a regular file, *aiobcp->aio_nbytes* is greater than 0, and the starting
4180 offset in *aiobcp->aio_offset* is before the end-of-file and is at or beyond the offset
4181 maximum in the open file description associated with *aiocbp->aio_fildes*.

4182 EXAMPLES

4183 None.

4184 APPLICATION USAGE

4185 The *aio_read()* function is part of the Asynchronous Input and Output option and need not be
4186 available on all implementations.

4187 RATIONALE

4188 None.

4189 FUTURE DIRECTIONS

4190 None.

4191 SEE ALSO

4192 *aio_cancel()*, *aio_error()*, *lio_listio()*, *aio_return()*, *aio_write()*, *close()*, *exec*, *exit()*, *fork()*, *lseek()*,
4193 *read()*, the Base Definitions volume of IEEE Std 1003.1-2001, <*aio.h*>

4194 CHANGE HISTORY

4195 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4196 Large File Summit extensions are added.

4197 Issue 6

4198 The [ENOSYS] error condition has been removed as stubs need not be provided if an
4199 implementation does not support the Asynchronous Input and Output option.

4200 The APPLICATION USAGE section is added.

4201 The following new requirements on POSIX implementations derive from alignment with the
4202 Single UNIX Specification:

- 4203 • In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file
4204 description. This change is to support large files.
- 4205 • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support
4206 large files.

4207 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/12 is applied, rewording the 2
4208 DESCRIPTION when prioritized I/O is supported to account for threads, and removing the 2
4209 words “to the calling process” in the RETURN VALUE section. 2

4210 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/13 is applied, updating the [EINVAL] 2
4211 error, so that detection of an [EINVAL] error for an invalid value of *aiocbp->aio_reqprio* is only 2
4212 required if the Prioritized Input and Output option is supported. 2

4213 NAME

4214 aio_return — retrieve return status of an asynchronous I/O operation (**REALTIME**)

4215 SYNOPSIS

4216 AIO #include <aio.h>

4217 ssize_t aio_return(struct aiocb *aiocbp);

4218

4219 DESCRIPTION

4220 The *aio_return()* function shall return the return status associated with the **aiocb** structure
4221 referenced by the *aiocbp* argument. The return status for an asynchronous I/O operation is the
4222 value that would be returned by the corresponding *read()*, *write()*, or *fsync()* function call. If the
4223 error status for the operation is equal to [EINPROGRESS], then the return status for the
4224 operation is undefined. The *aio_return()* function may be called exactly once to retrieve the
4225 return status of a given asynchronous operation; thereafter, if the same **aiocb** structure is used in
4226 a call to *aio_return()* or *aio_error()*, an error may be returned. When the **aiocb** structure referred
4227 to by *aiocbp* is used to submit another asynchronous operation, then *aio_return()* may be
4228 successfully used to retrieve the return status of that operation.

4229 RETURN VALUE

4230 If the asynchronous I/O operation has completed, then the return status, as described for *read()*,
4231 *write()*, and *fsync()*, shall be returned. If the asynchronous I/O operation has not yet completed,
4232 the results of *aio_return()* are undefined.

4233 ERRORS

4234 The *aio_return()* function may fail if:

4235 [EINVAL] The *aiocbp* argument does not refer to an asynchronous operation whose
4236 return status has not yet been retrieved.

4237 EXAMPLES

4238 None.

4239 APPLICATION USAGE

4240 The *aio_return()* function is part of the Asynchronous Input and Output option and need not be
4241 available on all implementations.

4242 RATIONALE

4243 None.

4244 FUTURE DIRECTIONS

4245 None.

4246 SEE ALSO

4247 *aio_cancel()*, *aio_error()*, *aio_fsync()*, *aio_read()*, *aio_write()*, *close()*, *exec*, *exit()*, *fork()*, *lio_listio()*,
4248 *lseek()*, *read()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**aio.h**>

4249 CHANGE HISTORY

4250 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4251 Issue 6

4252 The [ENOSYS] error condition has been removed as stubs need not be provided if an
4253 implementation does not support the Asynchronous Input and Output option.

4254 The APPLICATION USAGE section is added.

4255 The [EINVAL] error condition is made optional. This is for consistency with the DESCRIPTION.

4256 **NAME**4257 aio_suspend — wait for an asynchronous I/O request (**REALTIME**)4258 **SYNOPSIS**

```
4259 AIO #include <aio.h>
4260
4261     int aio_suspend(const struct aiocb * const list[], int nent,
4262                      const struct timespec *timeout);
```

4263 **DESCRIPTION**

4264 The *aio_suspend()* function shall suspend the calling thread until at least one of the asynchronous
 4265 I/O operations referenced by the *list* argument has completed, until a signal interrupts the
 4266 function, or, if *timeout* is not NULL, until the time interval specified by *timeout* has passed. If any
 4267 of the **aiocb** structures in the list correspond to completed asynchronous I/O operations (that is,
 4268 the error status for the operation is not equal to [EINPROGRESS]) at the time of the call, the
 4269 function shall return without suspending the calling thread. The *list* argument is an array of
 4270 pointers to asynchronous I/O control blocks. The *nent* argument indicates the number of
 4271 elements in the array. Each **aiocb** structure pointed to has been used in initiating an
 4272 asynchronous I/O request via *aio_read()*, *aio_write()*, or *lio_listio()*. This array may contain
 4273 NULL pointers, which are ignored. If this array contains pointers that refer to **aiocb** structures
 4274 that have not been used in submitting asynchronous I/O, the effect is undefined.

4275 If the time interval indicated in the **timespec** structure pointed to by *timeout* passes before any of
 4276 the I/O operations referenced by *list* are completed, then *aio_suspend()* shall return with an
 4277 error. If the Monotonic Clock option is supported, the clock that shall be used to measure this
 4278 time interval shall be the **CLOCK_MONOTONIC** clock.

4279 **RETURN VALUE**

4280 If the *aio_suspend()* function returns after one or more asynchronous I/O operations have
 4281 completed, the function shall return zero. Otherwise, the function shall return a value of -1 and
 4282 set *errno* to indicate the error.

4283 The application may determine which asynchronous I/O completed by scanning the associated
 4284 error and return status using *aio_error()* and *aio_return()*, respectively.

4285 **ERRORS**

4286 The *aio_suspend()* function shall fail if:

4287 [EAGAIN]	No asynchronous I/O indicated in the list referenced by <i>list</i> completed in the 4288 time interval indicated by <i>timeout</i> .
4289 [EINTR]	A signal interrupted the <i>aio_suspend()</i> function. Note that, since each 4290 asynchronous I/O operation may possibly provoke a signal when it 4291 completes, this error return may be caused by the completion of one (or more) 4292 of the very I/O operations being awaited.

4293 **EXAMPLES**

4294 None.

4295 **APPLICATION USAGE**

4296 The *aio_suspend()* function is part of the Asynchronous Input and Output option and need not
 4297 be available on all implementations.

4298 **RATIONALE**

4299 None.

4300 FUTURE DIRECTIONS

4301 None.

4302 SEE ALSO

4303 *aio_read()*, *aio_write()*, *lio_listio()*, the Base Definitions volume of IEEE Std 1003.1-2001, <aio.h>

4304 CHANGE HISTORY

4305 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4306 Issue 6

4307 The [ENOSYS] error condition has been removed as stubs need not be provided if an
4308 implementation does not support the Asynchronous Input and Output option.

4309 The APPLICATION USAGE section is added.

4310 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the
4311 CLOCK_MONOTONIC clock, if supported, is used.

4312 **NAME**4313 — asynchronous write to a file (**REALTIME**)4314 **SYNOPSIS**4315 AIO

```
#include <aio.h>
```

4316

```
int aio_write(struct aiocb *aiocbp);
```

4317

4318 **DESCRIPTION**

4319 The *aio_write()* function shall write *aiocbp->aio_nbytes* to the file associated with *aiocbp->aio_fildes*
 4320 from the buffer pointed to by *aiocbp->aio_buf*. The function shall return when the write request
 4321 has been initiated or, at a minimum, queued to the file or device.

4322 PIO If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted
 4323 at a priority equal to a base scheduling priority minus *aiocbp->aio_repprio*. If Thread Execution
 4324 Scheduling is not supported, then the base scheduling priority is that of the calling process;
 4325 PIO TPS otherwise, the base scheduling priority is that of the calling thread.

4326 The *aiocbp* argument may be used as an argument to *aio_error()* and *aio_return()* in order to
 4327 determine the error status and return status, respectively, of the asynchronous operation while it
 4328 is proceeding.

4329 The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp->aio_buf* or
 4330 the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O
 4331 completion, then the behavior is undefined.

4332 If O_APPEND is not set for the file descriptor *aio_fildes*, then the requested operation shall take
 4333 place at the absolute position in the file as given by *aio_offset*, as if *lseek()* were called
 4334 immediately prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to
 4335 SEEK_SET. If O_APPEND is set for the file descriptor, write operations append to the file in the
 4336 same order as the calls were made. After a successful call to enqueue an asynchronous I/O
 4337 operation, the value of the file offset for the file is unspecified.

4338 The *aiocbp->aio_lio_opcode* field shall be ignored by *aio_write()*.

4339 Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

4340 SIO If synchronized I/O is enabled on the file associated with *aiocbp->aio_fildes*, the behavior of this
 4341 function shall be according to the definitions of synchronized I/O data integrity completion, and
 4342 synchronized I/O file integrity completion.

4343 For any system action that changes the process memory space while an asynchronous I/O is
 4344 outstanding to the address range being changed, the result of that action is undefined.

4345 For regular files, no data transfer shall occur past the offset maximum established in the open
 4346 file description associated with *aiocbp->aio_fildes*.

4347 **RETURN VALUE**

4348 The *aio_write()* function shall return the value zero if the I/O operation is successfully queued; 2
 4349 otherwise, the function shall return the value -1 and set *errno* to indicate the error.

4350 **ERRORS**

4351 The *aio_write()* function shall fail if:

4352 [EAGAIN] The requested asynchronous I/O operation was not queued due to system
 4353 resource limitations.

4354 Each of the following conditions may be detected synchronously at the time of the call to
 4355 *aio_write()*, or asynchronously. If any of the conditions below are detected synchronously, the

4356 *aio_write()* function shall return `-1` and set *errno* to the corresponding value. If any of the
 4357 conditions below are detected asynchronously, the return status of the asynchronous operation
 4358 shall be set to `-1`, and the error status of the asynchronous operation is set to the corresponding
 4359 value.

4360 [EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for writing.

4361 [EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid,
 4362 *aiocbp->aio_reqprio* is not a valid value, or *aiocbp->aio_nbytes* is an invalid
 4363 value. 2
 2
 2

4364 In the case that the *aio_write()* successfully queues the I/O operation, the return status of the
 4365 asynchronous operation shall be one of the values normally returned by the *write()* function call.
 4366 If the operation is successfully queued but is subsequently canceled or encounters an error, the
 4367 error status for the asynchronous operation contains one of the values normally set by the
 4368 *write()* function call, or one of the following:

4369 [EBADF] The *aiocbp->aio_fildes* argument is not a valid file descriptor open for writing.

4370 [EINVAL] The file offset value implied by *aiocbp->aio_offset* would be invalid.

4371 [ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit
 4372 *aio_cancel()* request.

4373 The following condition may be detected synchronously or asynchronously:

4374 [EFBIG] The file is a regular file, *aiobcp->aio_nbytes* is greater than 0, and the starting
 4375 offset in *aiobcp->aio_offset* is at or beyond the offset maximum in the open file
 4376 description associated with *aiocbp->aio_fildes*. 2

4377 EXAMPLES

4378 None.

4379 APPLICATION USAGE

4380 The *aio_write()* function is part of the Asynchronous Input and Output option and need not be
 4381 available on all implementations.

4382 RATIONALE

4383 None.

4384 FUTURE DIRECTIONS

4385 None.

4386 SEE ALSO

4387 *aio_cancel()*, *aio_error()*, *aio_read()*, *aio_return()*, *close()*, *exec*, *exit()*, *fork()*, *lio_listio()*, *lseek()*,
 4388 *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <*aio.h*>

4389 CHANGE HISTORY

4390 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

4391 Large File Summit extensions are added.

4392 Issue 6

4393 The [ENOSYS] error condition has been removed as stubs need not be provided if an
 4394 implementation does not support the Asynchronous Input and Output option.

4395 The APPLICATION USAGE section is added.

4396 The following new requirements on POSIX implementations derive from alignment with the
 4397 Single UNIX Specification:

- 4398 • In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs
4399 past the offset maximum established in the open file description associated with
4400 *aiocbp->aio_fildes*.
4401 • The [EFBIG] error is added as part of the large file support extensions.
4402 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/14 is applied, rewording the 2
4403 DESCRIPTION when prioritized I/O is supported to account for threads, and removing the 2
4404 words “to the calling process” in the RETURN VALUE section. 2
4405 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/15 is applied, updating the [EINVAL] 2
4406 error, so that detection of an [EINVAL] error for an invalid value of *aiocbp->aio_reqprio* is only 2
4407 required if the Prioritized Input and Output option is supported. 2

4408 **NAME**

4409 alarm — schedule an alarm signal

4410 **SYNOPSIS**

4411 #include <unistd.h>
4412
4412 unsigned alarm(unsigned seconds);

4413 **DESCRIPTION**

4414 The *alarm()* function shall cause the system to generate a SIGALRM signal for the process after
4415 the number of realtime seconds specified by *seconds* have elapsed. Processor scheduling delays
4416 may prevent the process from handling the signal as soon as it is generated.

4417 If *seconds* is 0, a pending alarm request, if any, is canceled.

4418 Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner.
4419 If the SIGALRM signal has not yet been generated, the call shall result in rescheduling the time
4420 at which the SIGALRM signal is generated.

4421 XSI Interactions between *alarm()* and any of *setitimer()*, *ualarm()*, or *usleep()* are unspecified.

4422 **RETURN VALUE**

4423 If there is a previous *alarm()* request with time remaining, *alarm()* shall return a non-zero value
4424 that is the number of seconds until the previous request would have generated a SIGALRM
4425 signal. Otherwise, *alarm()* shall return 0.

4426 **ERRORS**

4427 The *alarm()* function is always successful, and no return value is reserved to indicate an error.

4428 **EXAMPLES**

4429 None.

4430 **APPLICATION USAGE**

4431 The *fork()* function clears pending alarms in the child process. A new process image created by
4432 one of the *exec* functions inherits the time left to an alarm signal in the old process' image.

4433 Application writers should note that the type of the argument *seconds* and the return value of
4434 *alarm()* is **unsigned**. That means that a Strictly Conforming POSIX System Interfaces
4435 Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX},
4436 which the ISO C standard sets as 65 535, and any application passing a larger value is restricting
4437 its portability. A different type was considered, but historical implementations, including those
4438 with a 16-bit **int** type, consistently use either **unsigned** or **int**.

4439 Application writers should be aware of possible interactions when the same process uses both
4440 the *alarm()* and *sleep()* functions.

4441 **RATIONALE**

4442 Many historical implementations (including Version 7 and System V) allow an alarm to occur up
4443 to a second early. Other implementations allow alarms up to half a second or one clock tick
4444 early or do not allow them to occur early at all. The latter is considered most appropriate, since it
4445 gives the most predictable behavior, especially since the signal can always be delayed for an
4446 indefinite amount of time due to scheduling. Applications can thus choose the *seconds* argument
4447 as the minimum amount of time they wish to have elapse before the signal.

4448 The term "realtime" here and elsewhere (*sleep()*, *times()*) is intended to mean "wall clock" time
4449 as common English usage, and has nothing to do with "realtime operating systems". It is in
4450 contrast to *virtual time*, which could be misinterpreted if just *time* were used.

4451 In some implementations, including 4.3 BSD, very large values of the *seconds* argument are
4452 silently rounded down to an implementation-specific maximum value. This maximum is large 2

4453 enough (to the order of several months) that the effect is not noticeable.

4454 There were two possible choices for alarm generation in multi-threaded applications: generation
4455 for the calling thread or generation for the process. The first option would not have been
4456 particularly useful since the alarm state is maintained on a per-process basis and the alarm that
4457 is established by the last invocation of *alarm()* is the only one that would be active.

4458 Furthermore, allowing generation of an asynchronous signal for a thread would have introduced
4459 an exception to the overall signal model. This requires a compelling reason in order to be
4460 justified.

4461 **FUTURE DIRECTIONS**

4462 None.

4463 **SEE ALSO**

4464 *alarm()*, *exec*, *fork()*, *getitimer()*, *pause()*, *sigaction()*, *sleep()*, *ualarm()*, *usleep()*, the Base
4465 Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>, <**unistd.h**>

4466 **CHANGE HISTORY**

4467 First released in Issue 1. Derived from Issue 1 of the SVID.

4468 **Issue 6**

4469 The following new requirements on POSIX implementations derive from alignment with the
4470 Single UNIX Specification:

- 4471 • The DESCRIPTION is updated to indicate that interactions with the *setitimer()*, *ualarm()*, and
4472 *usleep()* functions are unspecified.

4473 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/16 is applied, replacing “an 2
4474 implementation-defined maximum value” with “an implementation-specific maximum value” 2
4475 in the RATIONALE. 2

4476 NAME

4477 asctime, asctime_r — convert date and time to a string

4478 SYNOPSIS

```
4479 #include <time.h>
4480 char *asctime(const struct tm *timeptr);
4481 TSF char *asctime_r(const struct tm *restrict tm, char *restrict buf);
```

4483 DESCRIPTION

CX For *asctime()*: 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.

The *asctime()* function shall convert the broken-down time in the structure pointed to by *timeptr* into a string in the form:

```
4489 Sun Sep 16 01:03:52 1973\n\0
```

4490 using the equivalent of the following algorithm:

```
4491 char *asctime(const struct tm *timeptr)
4492 {
4493     static char wday_name[7][3] = {
4494         "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
4495     };
4496     static char mon_name[12][3] = {
4497         "Jan", "Feb", "Mar", "Apr", "May", "Jun",
4498         "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
4499     };
4500     static char result[26];
4501
4502     sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
4503             wday_name[timeptr->tm_wday],
4504             mon_name[timeptr->tm_mon],
4505             timeptr->tm_mday, timeptr->tm_hour,
4506             timeptr->tm_min, timeptr->tm_sec,
4507             1900 + timeptr->tm_year);
4508     return result;
4509 }
```

4509 The **tm** structure is defined in the **<time.h>** header.

CX 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.

4514 The *asctime()* function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *asctime_r()* function shall convert the broken-down time in the structure pointed to by *tm* into a string (of the same form as that returned by *asctime()*) that is placed in the user-supplied buffer pointed to by *buf* (which shall contain at least 26 bytes) and then return *buf*.

4519 RETURN VALUE

4520 CX Upon successful completion, *asctime*() shall return a pointer to the string. If the function is 2
4521 unsuccessful, it shall return NULL. 2

4522 TSF Upon successful completion, *asctime_r*() shall return a pointer to a character string containing
4523 the date and time. This string is pointed to by the argument *buf*. If the function is unsuccessful,
4524 it shall return NULL.

4525 ERRORS

4526 No errors are defined.

4527 EXAMPLES

4528 None.

4529 APPLICATION USAGE

4530 Values for the broken-down time structure can be obtained by calling *gmtime*() or *localtime*().
4531 This function is included for compatibility with older implementations, and does not support
4532 localized date and time formats. Applications should use *strftime*() to achieve maximum
4533 portability.

4534 The *asctime_r*() function is thread-safe and shall return values in a user-supplied buffer instead
4535 of possibly using a static data area that may be overwritten by each call.

4536 RATIONALE

4537 None.

4538 FUTURE DIRECTIONS

4539 None.

4540 SEE ALSO

4541 *clock*(), *ctime*(), *difftime*(), *gmtime*(), *localtime*(), *mktime*(), *strftime*(), *strptime*(), *time*(), *utime*(),
4542 the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

4543 CHANGE HISTORY

4544 First released in Issue 1. Derived from Issue 1 of the SVID.

4545 Issue 5

4546 Normative text previously in the APPLICATION USAGE section is moved to the
4547 DESCRIPTION.

4548 The *asctime_r*() function is included for alignment with the POSIX Threads Extension.

4549 A note indicating that the *asctime*() function need not be reentrant is added to the
4550 DESCRIPTION.

4551 Issue 6

4552 The *asctime_r*() function is marked as part of the Thread-Safe Functions option.

4553 Extensions beyond the ISO C standard are marked.

4554 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
4555 its avoidance of possibly using a static data area.

4556 The DESCRIPTION of *asctime_r*() is updated to describe the format of the string returned.

4557 The **restrict** keyword is added to the *asctime_r*() prototype for alignment with the
4558 ISO/IEC 9899:1999 standard.

4559	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/17 is applied, adding the CX extension in	2
4560	the RETURN VALUE section requiring that if the <i>asctime()</i> function is unsuccessful it returns	2
4561	NULL.	2

4562 **NAME**

4563 asin, asinf, asinl — arc sine function

4564 **SYNOPSIS**

```
4565     #include <math.h>
4566
4566     double asin(double x);
4567     float asinf(float x);
4568     long double asinl(long double x);
```

4569 **DESCRIPTION**

4570 CX 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.

4573 These functions shall compute the principal value of the arc sine of their argument x . The value of x should be in the range $[-1,1]$.

4575 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.

4579 **RETURN VALUE**

4580 Upon successful completion, these functions shall return the arc sine of x , in the range $[-\pi/2, \pi/2]$ radians.

4582 MX For finite values of x not in the range $[-1,1]$, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

4584 MX If x is NaN, a NaN shall be returned.

4585 If x is ± 0 , x shall be returned.

4586 If x is $\pm\text{Inf}$, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

4588 If x is subnormal, a range error may occur and x should be returned.

4589 **ERRORS**

4590 These functions shall fail if:

4591 MX Domain Error The x argument is finite and is not in the range $[-1,1]$, or is $\pm\text{Inf}$.

4592 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.

4596 These functions may fail if:

4597 MX Range Error The value of x is subnormal.

4598 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 underflow floating-point exception shall be raised.

4602 EXAMPLES

4603 None.

4604 APPLICATION USAGE

4605 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
4606 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4607 RATIONALE

4608 None.

4609 FUTURE DIRECTIONS

4610 None.

4611 SEE ALSO

4612 *feclearexcept()*, *fetestexcept()*, *isnan()*, *sin()*, the Base Definitions volume of IEEE Std 1003.1-2001,
4613 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

4614 CHANGE HISTORY

4615 First released in Issue 1. Derived from Issue 1 of the SVID.

4616 Issue 5

4617 The DESCRIPTION is updated to indicate how an application should check for an error. This
4618 text was previously published in the APPLICATION USAGE section.

4619 Issue 6

4620 The *asinf()* and *asinl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

4621 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
4622 revised to align with the ISO/IEC 9899:1999 standard.

4623 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
4624 marked.

4625 **NAME**

4626 asinh, asinhf, asinhl — inverse hyperbolic sine functions

4627 **SYNOPSIS**

```
4628     #include <math.h>
4629
4630     double asinh(double x);
4631     float asinhf(float x);
4632     long double asinhl(long double x);
```

4632 **DESCRIPTION**

4633 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4634 conflict between the requirements described here and the ISO C standard is unintentional. This
4635 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4636 These functions shall compute the inverse hyperbolic sine of their argument *x*.

4637 An application wishing to check for error situations should set *errno* to zero and call
4638 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
4639 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
4640 zero, an error has occurred.

4641 **RETURN VALUE**

4642 Upon successful completion, these functions shall return the inverse hyperbolic sine of their
4643 argument.

4644 MX If *x* is NaN, a NaN shall be returned.

4645 If *x* is ±0, or ±Inf, *x* shall be returned.

4646 If *x* is subnormal, a range error may occur and *x* should be returned.

4647 **ERRORS**

4648 These functions may fail if:

4649 MX Range Error The value of *x* is subnormal.

4650 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
4651 then *errno* shall be set to [ERANGE]. If the integer expression
4652 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
4653 floating-point exception shall be raised.

4654 **EXAMPLES**

4655 None.

4656 **APPLICATION USAGE**

4657 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
4658 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4659 **RATIONALE**

4660 None.

4661 **FUTURE DIRECTIONS**

4662 None.

4663 **SEE ALSO**

4664 *feclearexcept()*, *fetestexcept()*, *sinh()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section
4665 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

4666 CHANGE HISTORY

4667 First released in Issue 4, Version 2.

4668 Issue 5

4669 Moved from X/OPEN UNIX extension to BASE.

4670 Issue 6

4671 The *asinh()* function is no longer marked as an extension.

4672 The *asinhf()* and *asinhl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

4674 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

4675 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

4678 **NAME**
4679 `asinl` — arc sine function

4680 **SYNOPSIS**

4681 `#include <math.h>`
4682 `long double asinl(long double x);`

4683 **DESCRIPTION**

4684 Refer to `asin()`.

4685 NAME

4686 assert — insert program diagnostics

4687 SYNOPSIS

```
4688 #include <assert.h>
4689 void assert(scalar expression);
```

4690 DESCRIPTION

4691 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4692 conflict between the requirements described here and the ISO C standard is unintentional. This
4693 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4694 The **assert()** macro shall insert diagnostics into programs; it shall expand to a **void** expression.
4695 When it is executed, if *expression* (which shall have a **scalar** type) is false (that is, compares equal
4696 to 0), **assert()** shall write information about the particular call that failed on *stderr* and shall call
4697 **abort()**.

4698 The information written about the call that failed shall include the text of the argument, the
4699 name of the source file, the source file line number, and the name of the enclosing function; the
4700 latter are, respectively, the values of the preprocessing macros **_FILE_** and **_LINE_** and of
4701 the identifier **_func_**.

4702 Forcing a definition of the name **NDEBUG**, either from the compiler command line or with the
4703 preprocessor control statement **#define NDEBUG** ahead of the **#include <assert.h>** statement,
4704 shall stop assertions from being compiled into the program.

4705 RETURN VALUE

4706 The **assert()** macro shall not return a value.

4707 ERRORS

4708 No errors are defined.

4709 EXAMPLES

4710 None.

4711 APPLICATION USAGE

4712 None.

4713 RATIONALE

4714 None.

4715 FUTURE DIRECTIONS

4716 None.

4717 SEE ALSO

4718 **abort()**, *stderr*, the Base Definitions volume of IEEE Std 1003.1-2001, **<assert.h>**

4719 CHANGE HISTORY

4720 First released in Issue 1. Derived from Issue 1 of the SVID.

4721 Issue 6

4722 The prototype for the *expression* argument to **assert()** is changed from **int** to **scalar** for alignment
4723 with the ISO/IEC 9899:1999 standard.

4724 The DESCRIPTION of **assert()** is updated for alignment with the ISO/IEC 9899:1999 standard.

4725 NAME

4726 atan, atanf, atanl — arc tangent function

4727 SYNOPSIS

```
4728       #include <math.h>
4729
4730       double atan(double x);
4731       float atanf(float x);
4732       long double atanl(long double x);
```

4732 DESCRIPTION

4733 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4734 conflict between the requirements described here and the ISO C standard is unintentional. This
4735 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4736 These functions shall compute the principal value of the arc tangent of their argument *x*.

4737 An application wishing to check for error situations should set *errno* to zero and call
4738 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
4739 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
4740 zero, an error has occurred.

4741 RETURN VALUE

4742 Upon successful completion, these functions shall return the arc tangent of *x* in the range
4743 $[-\pi/2, \pi/2]$ radians.

4744 MX If *x* is NaN, a NaN shall be returned.

4745 If *x* is ± 0 , *x* shall be returned.

4746 If *x* is $\pm\text{Inf}$, $\pm\pi/2$ shall be returned.

4747 If *x* is subnormal, a range error may occur and *x* should be returned.

4748 ERRORS

4749 These functions may fail if:

4750 MX Range Error The value of *x* is subnormal.

4751 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
4752 then *errno* shall be set to [ERANGE]. If the integer expression
4753 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
4754 floating-point exception shall be raised.

4755 EXAMPLES

4756 None.

4757 APPLICATION USAGE

4758 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
4759 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

4760 RATIONALE

4761 None.

4762 FUTURE DIRECTIONS

4763 None.

4764 SEE ALSO

4765 *atan2()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *tan()*, the Base Definitions volume of
4766 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
4767 *<math.h>*

4768 **CHANGE HISTORY**

4769 First released in Issue 1. Derived from Issue 1 of the SVID.

4770 **Issue 5**

4771 The DESCRIPTION is updated to indicate how an application should check for an error. This
4772 text was previously published in the APPLICATION USAGE section.

4773 **Issue 6**

4774 The *atanf()* and *atanl()* functions are added for alignment with the ISO/IEC 9899: 1999 standard.

4775 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
4776 revised to align with the ISO/IEC 9899: 1999 standard.

4777 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are
4778 marked.

4779 **NAME**

4780 atan2, atan2f, atan2l — arc tangent functions

4781 **SYNOPSIS**

```
4782     #include <math.h>
4783
4784     double atan2(double y, double x);
4785     float atan2f(float y, float x);
4786     long double atan2l(long double y, long double x);
```

4786 **DESCRIPTION**

4787 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 4788 conflict between the requirements described here and the ISO C standard is unintentional. This
 4789 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4790 These functions shall compute the principal value of the arc tangent of y/x , using the signs of
 4791 both arguments to determine the quadrant of the return value.

4792 An application wishing to check for error situations should set *errno* to zero and call
 4793 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
 4794 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
 4795 zero, an error has occurred.

4796 **RETURN VALUE**

4797 Upon successful completion, these functions shall return the arc tangent of y/x in the range
 4798 $[-\pi, \pi]$ radians.

4799 If y is ± 0 and x is < 0 , $\pm\pi$ shall be returned.

4800 If y is ± 0 and x is > 0 , ± 0 shall be returned.

4801 If y is < 0 and x is ± 0 , $-\pi/2$ shall be returned.

4802 If y is > 0 and x is ± 0 , $\pi/2$ shall be returned.

4803 If x is 0, a pole error shall not occur.

4804 MX If either x or y is NaN, a NaN shall be returned.

4805 If the result underflows, a range error may occur and y/x should be returned.

4806 If y is ± 0 and x is -0 , $\pm\pi$ shall be returned.

4807 If y is ± 0 and x is $+0$, ± 0 shall be returned.

4808 For finite values of $\pm y > 0$, if x is $-\text{Inf}$, $\pm\pi$ shall be returned.

4809 For finite values of $\pm y > 0$, if x is $+\text{Inf}$, ± 0 shall be returned.

4810 For finite values of x , if y is $\pm\text{Inf}$, $\pm\pi/2$ shall be returned.

4811 If y is $\pm\text{Inf}$ and x is $-\text{Inf}$, $\pm 3\pi/4$ shall be returned.

4812 If y is $\pm\text{Inf}$ and x is $+\text{Inf}$, $\pm\pi/4$ shall be returned.

4813 If both arguments are 0, a domain error shall not occur.

4814 **ERRORS**

4815 These functions may fail if:

4816 MX Range Error The result underflows.

4817 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 4818 then *errno* shall be set to [ERANGE]. If the integer expression

4819 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
4820 floating-point exception shall be raised.

4821 EXAMPLES

4822 Converting Cartesian to Polar Coordinates System

4823 The function below uses `atan2()` to convert a 2d vector expressed in cartesian coordinates (x,y) to
4824 the polar coordinates $(rho,theta)$. There are other ways to compute the angle $theta$, using `asin()`
4825 `acos()`, or `atan()`. However, `atan2()` presents here two advantages:
22
22
22

- The angle's quadrant is automatically determined.
 - The singular cases $(0,y)$ are taken into account.

Finally, this example uses `hypot()` rather than `sqrt()` since it is better for special cases; see `hypot()` for more information.

```
4830 #include <math.h>
4831 void
4832 cartesian_to_polar(const double x, const double y,
4833                      double *rho, double *theta
4834                      )
4835 {
4836     *rho    = hypot (x,y); /* better than sqrt(x*x+y*y) */
4837     *theta = atan2 (y,x);
4838 }
```

4839 APPLICATION USAGE

4840 On error, the expressions (`math_errhandling` & `MATH_ERRNO`) and (`math_errhandling` &
4841 `MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

4842 RATIONALE

4843 None.

4844 FUTURE DIRECTIONS

4845 None.

4846 SEE ALSO

4847 `acos()`, `asin()`, `atan()`, `feclearexcept()`, `fetestexcept()`, `hypot()`, `isnan()`, `sqrt()`, `tan()`, the Base
4848 Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for
4849 Mathematical Functions, `<math.h>`

4850 CHANGE HISTORY

4851 First released in Issue 1. Derived from Issue 1 of the SVID.

4852 Issue 5

4853 The DESCRIPTION is updated to indicate how an application should check for an error. This
4854 text was previously published in the APPLICATION USAGE section.

4855 Issue 6

4856 The `atan2f()` and `atan2l()` functions are added for alignment with the ISO/IEC 9899:1999
4857 standard.

4858 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
4859 revised to align with the ISO/IEC 9899:1999 standard.

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

4862
4863IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/18 is applied, adding the example to the
EXAMPLES section.2
2

```
4864 NAME
4865     atanf — arc tangent function
4866 SYNOPSIS
4867     #include <math.h>
4868     float atanf(float x);
4869 DESCRIPTION
4870     Refer to atan().
```

4871 **NAME**4872 `atanh, atanhf, atanhl — inverse hyperbolic tangent functions`4873 **SYNOPSIS**

```
4874     #include <math.h>
4875
4876     double atanh(double x);
4877     float atanhf(float x);
4878     long double atanhl(long double x);
```

4878 **DESCRIPTION**

4879 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 4880 conflict between the requirements described here and the ISO C standard is unintentional. This
 4881 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4882 These functions shall compute the inverse hyperbolic tangent of their argument *x*.

4883 An application wishing to check for error situations should set *errno* to zero and call
 4884 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
 4885 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
 4886 zero, an error has occurred.

4887 **RETURN VALUE**

4888 Upon successful completion, these functions shall return the inverse hyperbolic tangent of their
 4889 argument.

4890 If *x* is ±1, a pole error shall occur, and *atanh()*, *atanhf()*, and *atanhl()* shall return the value of the
 4891 macro `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively, with the same sign as the
 4892 correct value of the function.

4893 MX For finite $|x|>1$, a domain error shall occur, and either a NaN (if supported), or an
 4894 implementation-defined value shall be returned.

4895 MX If *x* is NaN, a NaN shall be returned.

4896 If *x* is ±0, *x* shall be returned.

4897 If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-
 4898 defined value shall be returned.

4899 If *x* is subnormal, a range error may occur and *x* should be returned.

4900 **ERRORS**

4901 These functions shall fail if:

4902 MX Domain Error The *x* argument is finite and not in the range [−1,1], or is ±Inf.

4903 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 4904 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 4905 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 4906 shall be raised.

4907 MX Pole Error The *x* argument is ±1.

4908 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 4909 then *errno* shall be set to [ERANGE]. If the integer expression
 4910 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-
 4911 zero floating-point exception shall be raised.

4912	These functions may fail if:	
4913	MX	Range Error The value of x is subnormal.
4914	If the integer expression (<code>math_errhandling & MATH_ERRNO</code>) is non-zero, then <code>errno</code> shall be set to [ERANGE]. If the integer expression (<code>math_errhandling & MATH_ERREXCEPT</code>) is non-zero, then the underflow floating-point exception shall be raised.	
4915		
4916		
4917		
4918	EXAMPLES	
4919	None.	
4920	APPLICATION USAGE	
4921	On error, the expressions (<code>math_errhandling & MATH_ERRNO</code>) and (<code>math_errhandling & MATH_ERREXCEPT</code>) are independent of each other, but at least one of them must be non-zero.	
4922		
4923	RATIONALE	
4924	None.	
4925	FUTURE DIRECTIONS	
4926	None.	
4927	SEE ALSO	
4928	<code>feclearexcept()</code> , <code>fetestexcept()</code> , <code>tanh()</code> , the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, < math.h >	
4929		
4930	CHANGE HISTORY	
4931	First released in Issue 4, Version 2.	
4932	Issue 5	
4933	Moved from X/OPEN UNIX extension to BASE.	
4934	Issue 6	
4935	The <code>atanh()</code> function is no longer marked as an extension.	
4936	The <code>atanhf()</code> and <code>atanhl()</code> functions are added for alignment with the ISO/IEC 9899:1999 standard.	
4937		
4938	The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.	
4939		
4940	IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.	
4941		

```
4942 NAME
4943      atanl — arc tangent function

4944 SYNOPSIS
4945      #include <math.h>
4946      long double atanl(long double x);

4947 DESCRIPTION
4948      Refer to atan().
```

4949 NAME

4950 atexit — register a function to run at process termination

4951 SYNOPSIS

4952 #include <stdlib.h>
4953 int atexit(void (*func)(void));

4954 DESCRIPTION

4955 CX The functionality described on this reference page is aligned with the ISO C standard. Any
4956 conflict between the requirements described here and the ISO C standard is unintentional. This
4957 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

4958 The *atexit()* function shall register the function pointed to by *func*, to be called without
4959 arguments at normal program termination. At normal program termination, all functions
4960 registered by the *atexit()* function shall be called, in the reverse order of their registration, except
4961 that a function is called after any previously registered functions that had already been called at
4962 the time it was registered. Normal termination occurs either by a call to *exit()* or a return from
4963 *main()*.

4964 At least 32 functions can be registered with *atexit()*.

4965 CX After a successful call to any of the *exec* functions, any functions previously registered by *atexit()*
4966 shall no longer be registered.

4967 RETURN VALUE

4968 Upon successful completion, *atexit()* shall return 0; otherwise, it shall return a non-zero value.

4969 ERRORS

4970 No errors are defined.

4971 EXAMPLES

4972 None.

4973 APPLICATION USAGE

4974 The functions registered by a call to *atexit()* must return to ensure that all registered functions
4975 are called.

4976 The application should call *sysconf()* to obtain the value of {ATEXIT_MAX}, the number of
4977 functions that can be registered. There is no way for an application to tell how many functions
4978 have already been registered with *atexit()*.

4979 Since the behavior is undefined if the *exit()* function is called more than once, portable 2
4980 applications calling *atexit()* must ensure that the *exit()* function is not called at normal process 2
4981 termination when all functions registered by the *atexit()* function are called. 2

4982 All functions registered by the *atexit()* function are called at normal process termination, which 2
4983 occurs by a call to the *exit()* function or a return from *main()* or on the last thread termination, 2
4984 when the behavior is as if the implementation called *exit()* with a zero argument at thread 2
4985 termination time. 2

4986 If, at normal process termination, a function registered by the *atexit()* function is called and a 2
4987 portable application needs to stop further *exit()* processing, it must call the *_exit()* function or 2
4988 the *_Exit()* function or one of the functions which cause abnormal process termination. 2

4989 RATIONALE

4990 None.

4991 FUTURE DIRECTIONS

4992 None.

4993 SEE ALSO

4994 ***exit(), sysconf(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>***

4995 CHANGE HISTORY

4996 First released in Issue 4. Derived from the ANSI C standard.

4997 Issue 6

4998 Extensions beyond the ISO C standard are marked.

4999 The DESCRIPTION is updated for alignment with the ISO/IEC 9899: 1999 standard.

5000 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/19 is applied, adding further clarification 2
5001 to the APPLICATION USAGE section. 2

5002 NAME

5003 atof — convert a string to a double-precision number

5004 SYNOPSIS

5005 #include <stdlib.h>
5006
5006 double atof(const char *str);

5007 DESCRIPTION

5008 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5009 conflict between the requirements described here and the ISO C standard is unintentional. This
5010 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5011 The call *atof(str)* shall be equivalent to:

5012 strtod(str, (char **)NULL),

5013 except that the handling of errors may differ. If the value cannot be represented, the behavior is
5014 undefined.

5015 RETURN VALUE

5016 The *atof()* function shall return the converted value if the value can be represented.

5017 ERRORS

5018 No errors are defined.

5019 EXAMPLES

5020 None.

5021 APPLICATION USAGE

5022 The *atof()* function is subsumed by *strtod()* but is retained because it is used extensively in
5023 existing code. If the number is not known to be in range, *strtod()* should be used because *atof()* is
5024 not required to perform any error checking.

5025 RATIONALE

5026 None.

5027 FUTURE DIRECTIONS

5028 None.

5029 SEE ALSO

5030 *strtod()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

5031 CHANGE HISTORY

5032 First released in Issue 1. Derived from Issue 1 of the SVID.

5033 NAME

5034 atoi — convert a string to an integer

5035 SYNOPSIS

```
5036     #include <stdlib.h>
5037
      int atoi(const char *str);
```

5038 DESCRIPTION

5039 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5040 conflict between the requirements described here and the ISO C standard is unintentional. This
5041 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5042 The call *atoi(str)* shall be equivalent to:

```
5043     (int) strtol(str, (char **)NULL, 10)
```

5044 except that the handling of errors may differ. If the value cannot be represented, the behavior is
5045 undefined.

5046 RETURN VALUE

5047 The *atoi()* function shall return the converted value if the value can be represented.

5048 ERRORS

5049 No errors are defined.

5050 EXAMPLES**5051 Converting an Argument**

5052 The following example checks for proper usage of the program. If there is an argument and the
5053 decimal conversion of this argument (obtained using *atoi()*) is greater than 0, then the program
5054 has a valid number of minutes to wait for an event.

```
5055     #include <stdlib.h>
5056     #include <stdio.h>
5057     ...
5058     int minutes_to_event;
5059     ...
5060     if (argc < 2 || ((minutes_to_event = atoi (argv[1]))) <= 0) {
5061         fprintf(stderr, "Usage: %s minutes\n", argv[0]); exit(1);
5062     }
5063     ...
```

5064 APPLICATION USAGE

5065 The *atoi()* function is subsumed by *strtol()* but is retained because it is used extensively in
5066 existing code. If the number is not known to be in range, *strtol()* should be used because *atoi()* is
5067 not required to perform any error checking.

5068 RATIONALE

5069 None.

5070 FUTURE DIRECTIONS

5071 None.

5072 SEE ALSO

5073 *strtol()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

5074 CHANGE HISTORY

5075 First released in Issue 1. Derived from Issue 1 of the SVID.

5076 NAME

5077 atol, atoll — convert a string to a long integer

5078 SYNOPSIS

```
5079     #include <stdlib.h>
5080
5081     long atol(const char *str);
5082     long long atoll(const char *nptr);
```

5082 DESCRIPTION

5083 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5084 conflict between the requirements described here and the ISO C standard is unintentional. This
5085 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5086 The call *atol(str)* shall be equivalent to:

```
5087     strtol(str, (char **)NULL, 10)
```

5088 The call *atoll(str)* shall be equivalent to:

```
5089     strtoll(nptr, (char **)NULL, 10)
```

5090 except that the handling of errors may differ. If the value cannot be represented, the behavior is
5091 undefined.

5092 RETURN VALUE

5093 These functions shall return the converted value if the value can be represented.

5094 ERRORS

5095 No errors are defined.

5096 EXAMPLES

5097 None.

5098 APPLICATION USAGE

5099 The *atol()* function is subsumed by *strtol()* but is retained because it is used extensively in
5100 existing code. If the number is not known to be in range, *strtol()* should be used because *atol()* is
5101 not required to perform any error checking.

5102 RATIONALE

5103 None.

5104 FUTURE DIRECTIONS

5105 None.

5106 SEE ALSO

5107 *strtol()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

5108 CHANGE HISTORY

5109 First released in Issue 1. Derived from Issue 1 of the SVID.

5110 Issue 6

5111 The *atoll()* function is added for alignment with the ISO/IEC 9899:1999 standard.

5112 NAME

5113 basename — return the last component of a pathname

5114 SYNOPSIS

5115 XSI #include <libgen.h>

5116 char *basename(char *path);

5117

5118 DESCRIPTION

5119 The *basename()* function shall take the pathname pointed to by *path* and return a pointer to the
5120 final component of the pathname, deleting any trailing ' / ' characters.

5121 If the string pointed to by *path* consists entirely of the ' / ' character, *basename()* shall return a 2
5122 pointer to the string " / ". If the string pointed to by *path* is exactly " / ", it is implementation- 2
5123 defined whether ' / ' or " / " is returned.

5124 If *path* is a null pointer or points to an empty string, *basename()* shall return a pointer to the
5125 string " . ".

5126 The *basename()* function may modify the string pointed to by *path*, and may return a pointer to
5127 static storage that may then be overwritten by a subsequent call to *basename()*.

5128 The *basename()* function need not be reentrant. A function that is not required to be reentrant is
5129 not required to be thread-safe.

5130 RETURN VALUE

5131 The *basename()* function shall return a pointer to the final component of *path*.

5132 ERRORS

5133 No errors are defined.

5134 EXAMPLES**5135 Using basename()**

5136 The following program fragment returns a pointer to the value *lib*, which is the base name of
5137 */usr/lib*.

```
5138 #include <libgen.h>
5139 ...
5140 char *name = "/usr/lib";
5141 char *base;
5142 base = basename(name);
5143 ...
```

5144 Sample Input and Output Strings for basename()

5145 In the following table, the input string is the value pointed to by *path*, and the output string is
5146 the return value of the *basename()* function.

5147	Input String	Output String
5148	" /usr/lib"	" lib"
5149	" /usr/ "	" usr"
5150	" / "	" / "
5151	" /// "	" / "
5152	" /usr//lib// "	" lib"

5153 APPLICATION USAGE

5154 None.

5155 RATIONALE

5156 None.

5157 FUTURE DIRECTIONS

5158 None.

5159 SEE ALSO

5160 *dirname()*, the Base Definitions volume of IEEE Std 1003.1-2001, <libgen.h>, the Shell and
5161 Utilities volume of IEEE Std 1003.1-2001, *basename*

5162 CHANGE HISTORY

5163 First released in Issue 4, Version 2.

5164 Issue 5

5165 Moved from X/OPEN UNIX extension to BASE.

5166 Normative text previously in the APPLICATION USAGE section is moved to the
5167 DESCRIPTION.

5168 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

5169 Issue 6

5170 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

5171 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/20 is applied, changing the 2
5172 DESCRIPTION to make it clear that the string referenced is the string pointed to by *path*. 2

5173 NAME

5174 **bcmcmp** — memory operations (**LEGACY**)

5175 SYNOPSIS

5176 XSI `#include <strings.h>`

5177 `int bcmcmp(const void *s1, const void *s2, size_t n);`

5178

5179 DESCRIPTION

5180 The *bcmcmp()* function shall compare the first *n* bytes of the area pointed to by *s1* with the area
5181 pointed to by *s2*.

5182 RETURN VALUE

5183 The *bcmcmp()* function shall return 0 if *s1* and *s2* are identical; otherwise, it shall return non-zero.
5184 Both areas are assumed to be *n* bytes long. If the value of *n* is 0, *bcmcmp()* shall return 0.

5185 ERRORS

5186 No errors are defined.

5187 EXAMPLES

5188 None.

5189 APPLICATION USAGE

5190 The *memcmp()* function is preferred over this function.

5191 For maximum portability, it is recommended to replace the function call to *bcmcmp()* as follows:

5192 `#define bcmcmp(b1,b2,len) memcmp((b1), (b2), (size_t)(len))`

5193 RATIONALE

5194 None.

5195 FUTURE DIRECTIONS

5196 This function may be withdrawn in a future version.

5197 SEE ALSO

5198 *memcmp()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<strings.h>**

5199 CHANGE HISTORY

5200 First released in Issue 4, Version 2.

5201 Issue 5

5202 Moved from X/OPEN UNIX extension to BASE.

5203 Issue 6

5204 This function is marked LEGACY.

5205 NAME

5206 **bcopy** — memory operations (LEGACY)

5207 SYNOPSIS

5208 XSI `#include <strings.h>`

5209 `void bcopy(const void *s1, void *s2, size_t n);`

5210

5211 DESCRIPTION

5212 The *bcopy()* function shall copy *n* bytes from the area pointed to by *s1* to the area pointed to by *s2*.

5214 The bytes are copied correctly even if the area pointed to by *s1* overlaps the area pointed to by *s2*.

5216 RETURN VALUE

5217 The *bcopy()* function shall not return a value.

5218 ERRORS

5219 No errors are defined.

5220 EXAMPLES

5221 None.

5222 APPLICATION USAGE

5223 The *memmove()* function is preferred over this function.

5224 The following are approximately equivalent (note the order of the arguments):

5225 `bcopy(s1,s2,n) ~= memmove(s2,s1,n)`

5226 For maximum portability, it is recommended to replace the function call to *bcopy()* as follows:

5227 `#define bcopy(b1,b2,len) (memmove((b2), (b1), (len)), (void) 0)`

5228 RATIONALE

5229 None.

5230 FUTURE DIRECTIONS

5231 This function may be withdrawn in a future version.

5232 SEE ALSO

5233 *memmove()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**strings.h**>

5234 CHANGE HISTORY

5235 First released in Issue 4, Version 2.

5236 Issue 5

5237 Moved from X/OPEN UNIX extension to BASE.

5238 Issue 6

5239 This function is marked LEGACY.

5240 **NAME**

5241 bind — bind a name to a socket

5242 **SYNOPSIS**

```
5243 #include <sys/socket.h>
5244 int bind(int socket, const struct sockaddr *address,
5245         socklen_t address_len);
```

5246 **DESCRIPTION**

5247 The *bind()* function shall assign a local socket address *address* to a socket identified by descriptor *socket* that has no local socket address assigned. Sockets created with the *socket()* function are initially unnamed; they are identified only by their address family.

5250 The *bind()* function takes the following arguments:

5251 <i>socket</i>	Specifies the file descriptor of the socket to be bound.
5252 <i>address</i>	Points to a sockaddr structure containing the address to be bound to the socket. The length and format of the address depend on the address family of the socket.
5255 <i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> argument.

5257 The socket specified by *socket* may require the process to have appropriate privileges to use the *bind()* function.

5259 **RETURN VALUE**

5260 Upon successful completion, *bind()* shall return 0; otherwise, -1 shall be returned and *errno* set to indicate the error.

5262 **ERRORS**

5263 The *bind()* function shall fail if:

5264 [EADDRINUSE] The specified address is already in use.

5265 [EADDRNOTAVAIL]

5266 The specified address is not available from the local machine.

5267 [EAFNOSUPPORT]

5268 The specified address is not a valid address for the address family of the
5269 specified socket.

5270 [EBADF] The *socket* argument is not a valid file descriptor.

5271 [EINVAL] The socket is already bound to an address, and the protocol does not support
5272 binding to a new address; or the socket has been shut down.

5273 [ENOTSOCK] The *socket* argument does not refer to a socket.

5274 [EOPNOTSUPP] The socket type of the specified socket does not support binding to an
5275 address.

5276 If the address family of the socket is AF_UNIX, then *bind()* shall fail if:

5277 [EACCES] A component of the path prefix denies search permission, or the requested
5278 name requires writing in a directory with a mode that denies write
5279 permission.

5280 [EDESTADDRREQ] or [EISDIR]

5281 The *address* argument is a null pointer.

5282	[EIO]	An I/O error occurred.
5283	[ELOOP]	A loop exists in symbolic links encountered during resolution of the pathname in <i>address</i> .
5284	[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
5285	[ENOENT]	A component of the pathname does not name an existing file or the pathname is an empty string.
5286	[ENOTDIR]	A component of the path prefix of the pathname in <i>address</i> is not a directory.
5287	[EROFS]	The name would reside on a read-only file system.
5288		The <i>bind()</i> function may fail if:
5289	[EACCES]	The specified address is protected and the current user does not have permission to bind to it.
5290	[EINVAL]	The <i>address_len</i> argument is not a valid length for the address family.
5291	[EISCONN]	The socket is already connected.
5292	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in <i>address</i> .
5293	[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
5294		
5295	[ENOBUFS]	Insufficient resources were available to complete the call.

5303 EXAMPLES

5304 None.

5305 APPLICATION USAGE

5306 An application program can retrieve the assigned socket name with the *getsockname()* function.

5307 RATIONALE

5308 None.

5309 FUTURE DIRECTIONS

5310 None.

5311 SEE ALSO

5312 *connect()*, *getsockname()*, *listen()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001,
5313 <sys/socket.h>

5314 CHANGE HISTORY

5315 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

5316 NAME

5317 **bsd_signal** — simplified signal facilities

5318 SYNOPSIS

5319 OB XSI **#include <signal.h>**
5320 **void (*bsd_signal(int sig, void (*func)(int)))(int);**
5321

5322 DESCRIPTION

5323 The *bsd_signal()* function provides a partially compatible interface for programs written to
5324 historical system interfaces (see APPLICATION USAGE).

5325 The function call *bsd_signal(sig, func)* shall be equivalent to the following:

```
5326       void (*bsd_signal(int sig, void (*func)(int)))(int)
5327       {
5328           struct sigaction act, oact;
5329           act.sa_handler = func;
5330           act.sa_flags = SA_RESTART;
5331           sigemptyset(&act.sa_mask);
5332           sigaddset(&act.sa_mask, sig);
5333           if (sigaction(sig, &act, &oact) == -1)
5334              return(SIG_ERR);
5335           return(oact.sa_handler);
5336       }
```

5337 The handler function should be declared:

```
5338       void handler(int sig);
```

5339 where *sig* is the signal number. The behavior is undefined if *func* is a function that takes more
5340 than one argument, or an argument of a different type.

5341 RETURN VALUE

5342 Upon successful completion, *bsd_signal()* shall return the previous action for *sig*. Otherwise,
5343 *SIG_ERR* shall be returned and *errno* shall be set to indicate the error.

5344 ERRORS

5345 Refer to *sigaction()*.

5346 EXAMPLES

5347 None.

5348 APPLICATION USAGE

5349 This function is a direct replacement for the BSD *signal()* function for simple applications that
5350 are installing a single-argument signal handler function. If a BSD signal handler function is being
5351 installed that expects more than one argument, the application has to be modified to use
5352 *sigaction()*. The *bsd_signal()* function differs from *signal()* in that the *SA_RESTART* flag is set
5353 and the *SA_RESETHAND* is clear when *bsd_signal()* is used. The state of these flags is not
5354 specified for *signal()*.

5355 It is recommended that new applications use the *sigaction()* function.

5356 RATIONALE

5357 None.

5358 FUTURE DIRECTIONS

5359 None.

5360 SEE ALSO

5361 *sigaction()*, *sigaddset()*, *sigemptyset()*, *signal()*, the Base Definitions volume of
5362 IEEE Std 1003.1-2001, <**signal.h**>

5363 CHANGE HISTORY

5364 First released in Issue 4, Version 2.

5365 Issue 5

5366 Moved from X/OPEN UNIX extension to BASE.

5367 Issue 6

5368 This function is marked obsolescent.

5369 **NAME**5370 **bsearch** — binary search a sorted table5371 **SYNOPSIS**

```
5372     #include <stdlib.h>
5373
5374     void *bsearch(const void *key, const void *base, size_t nel,
5375                   size_t width, int (*compar)(const void *, const void *));
```

5375 **DESCRIPTION**

5376 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 5377 conflict between the requirements described here and the ISO C standard is unintentional. This
 5378 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5379 The *bsearch()* function shall search an array of *nel* objects, the initial element of which is pointed
 5380 to by *base*, for an element that matches the object pointed to by *key*. The size of each element in
 5381 the array is specified by *width*. If the *nel* argument has the value zero, the comparison function
 5382 pointed to by *compar* shall not be called and no match shall be found.

5383 The comparison function pointed to by *compar* shall be called with two arguments that point to
 5384 the *key* object and to an array element, in that order.

5385 The application shall ensure that the comparison function pointed to by *compar* does not alter the
 5386 contents of the array. The implementation may reorder elements of the array between calls to the
 5387 comparison function, but shall not alter the contents of any individual element.

5388 The implementation shall ensure that the first argument is always a pointer to the key.

5389 When the same objects (consisting of *width* bytes, irrespective of their current positions in the
 5390 array) are passed more than once to the comparison function, the results shall be consistent with
 5391 one another. That is, the same object shall always compare the same way with the key.

5392 The application shall ensure that the function returns an integer less than, equal to, or greater
 5393 than 0 if the *key* object is considered, respectively, to be less than, to match, or to be greater than
 5394 the array element. The application shall ensure that the array consists of all the elements that
 5395 compare less than, all the elements that compare equal to, and all the elements that compare
 5396 greater than the *key* object, in that order.

5397 **RETURN VALUE**

5398 The *bsearch()* function shall return a pointer to a matching member of the array, or a null pointer
 5399 if no match is found. If two or more members compare equal, which member is returned is
 5400 unspecified.

5401 **ERRORS**

5402 No errors are defined.

5403 **EXAMPLES**

5404 The example below searches a table containing pointers to nodes consisting of a string and its
 5405 length. The table is ordered alphabetically on the string in the node pointed to by each entry.

5406 The code fragment below reads in strings and either finds the corresponding node and prints out
 5407 the string and its length, or prints an error message.

```
5408 #include <stdio.h>
5409 #include <stdlib.h>
5410 #include <string.h>
5411
5412 #define TABSIZE    1000
```

```
5412     struct node {                                /* These are stored in the table. */
5413         char *string;
5414         int length;
5415     };
5416     struct node table[TABSIZE];      /* Table to be searched. */
5417     .
5418     .
5419     .
5420 {
5421     struct node *node_ptr, node;
5422     /* Routine to compare 2 nodes. */
5423     int node_compare(const void *, const void *);
5424     char str_space[20];    /* Space to read string into. */
5425     .
5426     .
5427     .
5428     node.string = str_space;
5429     while (scanf("%s", node.string) != EOF) {
5430         node_ptr = (struct node *)bsearch((void *)(&node),
5431                                         (void *)table, TABSIZE,
5432                                         sizeof(struct node), node_compare);
5433         if (node_ptr != NULL) {
5434             (void)printf("string = %20s, length = %d\n",
5435                         node_ptr->string, node_ptr->length);
5436         } else {
5437             (void)printf("not found: %s\n", node.string);
5438         }
5439     }
5440 }
5441 /*
5442     This routine compares two nodes based on an
5443     alphabetical ordering of the string field.
5444 */
5445 int
5446 node_compare(const void *node1, const void *node2)
5447 {
5448     return strcoll(((const struct node *)node1)->string,
5449                    ((const struct node *)node2)->string);
5450 }
```

5451 APPLICATION USAGE

5452 The pointers to the key and the element at the base of the table should be of type pointer-to-
5453 element.

5454 The comparison function need not compare every byte, so arbitrary data may be contained in
5455 the elements in addition to the values being compared.

5456 In practice, the array is usually sorted according to the comparison function.

5457 RATIONALE

5458 The requirement that the second argument (hereafter referred to as *p*) to the comparison 1
5459 function is a pointer to an element of the array implies that for every call all of the following 1
5460 expressions are non-zero: 1

5461 ((char *)p - (char *)base) % width == 0
5462 (char *)p >= (char *)base
5463 (char *)p < (char *)base + nel * width

1
1
1

5464 FUTURE DIRECTIONS

5465 None.

5466 SEE ALSO

5467 *hcreate()*, *lsearch()*, *qsort()*, *tsearch()*, the Base Definitions volume of IEEE Std 1003.1-2001,
5468 *<stdlib.h>*

5469 CHANGE HISTORY

5470 First released in Issue 1. Derived from Issue 1 of the SVID.

5471 Issue 6

5472 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

5473 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/11 is applied, adding to the 1
5474 DESCRIPTION the last sentence of the first non-shaded paragraph, and the following three 1
5475 paragraphs. The RATIONALE section is also updated. These changes are for alignment with the 1
5476 ISO C standard. 1

5477 NAME

5478 btowc — single byte to wide character conversion

5479 SYNOPSIS

```
5480       #include <stdio.h>
5481       #include <wchar.h>
5482       wint_t btowc(int c);
```

5483 DESCRIPTION

5484 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5485 conflict between the requirements described here and the ISO C standard is unintentional. This
5486 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5487 The *btowc()* function shall determine whether *c* constitutes a valid (one-byte) character in the
5488 initial shift state.

5489 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

5490 RETURN VALUE

5491 The *btowc()* function shall return WEOF if *c* has the value EOF or if (**unsigned char**) *c* does not
5492 constitute a valid (one-byte) character in the initial shift state. Otherwise, it shall return the
5493 wide-character representation of that character.

5494 ERRORS

5495 No errors are defined.

5496 EXAMPLES

5497 None.

5498 APPLICATION USAGE

5499 None.

5500 RATIONALE

5501 None.

5502 FUTURE DIRECTIONS

5503 None.

5504 SEE ALSO

5505 *wctob()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

5506 CHANGE HISTORY

5507 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
5508 (E).

5509 NAME

5510 **bzero** — memory operations (**LEGACY**)

5511 SYNOPSIS

5512 XSI `#include <strings.h>`

5513 `void bzero(void *s, size_t n);`

5514

5515 DESCRIPTION

5516 The *bzero()* function shall place *n* zero-valued bytes in the area pointed to by *s*.

5517 RETURN VALUE

5518 The *bzero()* function shall not return a value.

5519 ERRORS

5520 No errors are defined.

5521 EXAMPLES

5522 None.

5523 APPLICATION USAGE

5524 The *memset()* function is preferred over this function.

5525 For maximum portability, it is recommended to replace the function call to *bzero()* as follows:

5526 `#define bzero(b,len) (memset((b), '\0', (len)), (void) 0)`

5527 RATIONALE

5528 None.

5529 FUTURE DIRECTIONS

5530 This function may be withdrawn in a future version.

5531 SEE ALSO

5532 *memset()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<strings.h>**

5533 CHANGE HISTORY

5534 First released in Issue 4, Version 2.

5535 Issue 5

5536 Moved from X/OPEN UNIX extension to BASE.

5537 Issue 6

5538 This function is marked LEGACY.

5539 NAME

5540 cabs, cabsf, cabsl — return a complex absolute value

5541 SYNOPSIS

```
5542     #include <complex.h>
5543
5544     double cabs(double complex z);
5545     float cabsf(float complex z);
5546     long double cabsl(long double complex z);
```

5546 DESCRIPTION

5547 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5548 conflict between the requirements described here and the ISO C standard is unintentional. This
5549 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5550 These functions shall compute the complex absolute value (also called norm, modulus, or
5551 magnitude) of *z*.

5552 RETURN VALUE

5553 These functions shall return the complex absolute value.

5554 ERRORS

5555 No errors are defined.

5556 EXAMPLES

5557 None.

5558 APPLICATION USAGE

5559 None.

5560 RATIONALE

5561 None.

5562 FUTURE DIRECTIONS

5563 None.

5564 SEE ALSO

5565 The Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

5566 CHANGE HISTORY

5567 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

5568 NAME

5569 cacos, cacosf, cacosl — complex arc cosine functions

5570 SYNOPSIS

```
5571     #include <complex.h>
5572
5573     double complex cacos(double complex z);
5574     float complex cacosf(float complex z);
5575     long double complex cacosl(long double complex z);
```

5575 DESCRIPTION

5576 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5577 conflict between the requirements described here and the ISO C standard is unintentional. This
5578 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5579 These functions shall compute the complex arc cosine of z , with branch cuts outside the interval
5580 $[-1, +1]$ along the real axis.

5581 RETURN VALUE

5582 These functions shall return the complex arc cosine value, in the range of a strip mathematically
5583 unbounded along the imaginary axis and in the interval $[0, \pi]$ along the real axis.

5584 ERRORS

5585 No errors are defined.

5586 EXAMPLES

5587 None.

5588 APPLICATION USAGE

5589 None.

5590 RATIONALE

5591 None.

5592 FUTURE DIRECTIONS

5593 None.

5594 SEE ALSO

5595 `ccos()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<complex.h>`

5596 CHANGE HISTORY

5597 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5598 NAME

5599 cacosh, cacoshf, cacoshl — complex arc hyperbolic cosine functions

5600 SYNOPSIS

```
5601       #include <complex.h>
5602
5603       double complex cacosh(double complex z);
5604       float complex cacoshf(float complex z);
5605       long double complex cacoshl(long double complex z);
```

5605 DESCRIPTION

5606 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5607 conflict between the requirements described here and the ISO C standard is unintentional. This
5608 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5609 These functions shall compute the complex arc hyperbolic cosine of z , with a branch cut at
5610 values less than 1 along the real axis.

5611 RETURN VALUE

5612 These functions shall return the complex arc hyperbolic cosine value, in the range of a half-strip
5613 of non-negative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

5614 ERRORS

5615 No errors are defined.

5616 EXAMPLES

5617 None.

5618 APPLICATION USAGE

5619 None.

5620 RATIONALE

5621 None.

5622 FUTURE DIRECTIONS

5623 None.

5624 SEE ALSO

5625 `ccosh()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<complex.h>`

5626 CHANGE HISTORY

5627 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

```
5628 NAME
5629      cacosl — complex arc cosine functions
5630 SYNOPSIS
5631      #include <complex.h>
5632      long double complex cacosl(long double complex z);
5633 DESCRIPTION
5634      Refer to cacos().
```

5635 NAME

5636 calloc — a memory allocator

5637 SYNOPSIS

```
5638       #include <stdlib.h>
5639       void *calloc(size_t nelem, size_t elsize);
```

5640 DESCRIPTION

5641 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5642 conflict between the requirements described here and the ISO C standard is unintentional. This
5643 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5644 The *calloc()* function shall allocate unused space for an array of *nelem* elements each of whose
5645 size in bytes is *elsize*. The space shall be initialized to all bits 0.

5646 The order and contiguity of storage allocated by successive calls to *calloc()* is unspecified. The
5647 pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to
5648 a pointer to any type of object and then used to access such an object or an array of such objects
5649 in the space allocated (until the space is explicitly freed or reallocated). Each such allocation
5650 shall yield a pointer to an object disjoint from any other object. The pointer returned shall point
5651 to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null
5652 pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-
5653 defined: the value returned shall be either a null pointer or a unique pointer.

5654 RETURN VALUE

5655 Upon successful completion with both *nelem* and *elsize* non-zero, *calloc()* shall return a pointer to
5656 the allocated space. If either *nelem* or *elsize* is 0, then either a null pointer or a unique pointer
5657 value that can be successfully passed to *free()* shall be returned. Otherwise, it shall return a null
5658 CX pointer and set *errno* to indicate the error.

5659 ERRORS

5660 The *calloc()* function shall fail if:

5661 CX [ENOMEM] Insufficient memory is available.

5662 EXAMPLES

5663 None.

5664 APPLICATION USAGE

5665 There is now no requirement for the implementation to support the inclusion of <**malloc.h**>.

5666 RATIONALE

5667 None.

5668 FUTURE DIRECTIONS

5669 None.

5670 SEE ALSO

5671 *free()*, *malloc()*, *realloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**stdlib.h**>

5672 CHANGE HISTORY

5673 First released in Issue 1. Derived from Issue 1 of the SVID.

5674 Issue 6

5675 Extensions beyond the ISO C standard are marked.

5676 The following new requirements on POSIX implementations derive from alignment with the
5677 Single UNIX Specification:

- 5678 • The setting of *errno* and the [ENOMEM] error condition are mandatory if an insufficient
5679 memory condition occurs.

5680 NAME

5681 carg, cargf, cargl — complex argument functions

5682 SYNOPSIS

```
5683 #include <complex.h>
5684 double carg(double complex z);
5685 float cargf(float complex z);
5686 long double cargl(long double complex z);
```

5687 DESCRIPTION

5688 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5689 conflict between the requirements described here and the ISO C standard is unintentional. This
5690 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5691 These functions shall compute the argument (also called phase angle) of z , with a branch cut
5692 along the negative real axis.

5693 RETURN VALUE

5694 These functions shall return the value of the argument in the interval $[-\pi, +\pi]$.

5695 ERRORS

5696 No errors are defined.

5697 EXAMPLES

5698 None.

5699 APPLICATION USAGE

5700 None.

5701 RATIONALE

5702 None.

5703 FUTURE DIRECTIONS

5704 None.

5705 SEE ALSO

5706 *cimag()*, *conj()*, *cproj()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

5707 CHANGE HISTORY

5708 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

5709 NAME

5710 casin, casinf, casinl — complex arc sine functions

5711 SYNOPSIS

```
5712       #include <complex.h>
5713       double complex casin(double complex z);
5714       float complex casinf(float complex z);
5715       long double complex casinl(long double complex z);
```

5716 DESCRIPTION

5717 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5718 conflict between the requirements described here and the ISO C standard is unintentional. This
5719 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5720 These functions shall compute the complex arc sine of z , with branch cuts outside the interval
5721 $[-1, +1]$ along the real axis.

5722 RETURN VALUE

5723 These functions shall return the complex arc sine value, in the range of a strip mathematically
5724 unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

5725 ERRORS

5726 No errors are defined.

5727 EXAMPLES

5728 None.

5729 APPLICATION USAGE

5730 None.

5731 RATIONALE

5732 None.

5733 FUTURE DIRECTIONS

5734 None.

5735 SEE ALSO

5736 `csin()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<complex.h>`

5737 CHANGE HISTORY

5738 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5739 NAME

5740 casinh, casinhf, casinhl — complex arc hyperbolic sine functions

5741 SYNOPSIS

```
5742 #include <complex.h>
5743 double complex casinh(double complex z);
5744 float complex casinhf(float complex z);
5745 long double complex casinhl(long double complex z);
```

5746 DESCRIPTION

5747 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5748 conflict between the requirements described here and the ISO C standard is unintentional. This
5749 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5750 These functions shall compute the complex arc hyperbolic sine of z , with branch cuts outside the
5751 interval $[-i, +i]$ along the imaginary axis.

5752 RETURN VALUE

5753 These functions shall return the complex arc hyperbolic sine value, in the range of a strip
5754 mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the
5755 imaginary axis.

5756 ERRORS

5757 No errors are defined.

5758 EXAMPLES

5759 None.

5760 APPLICATION USAGE

5761 None.

5762 RATIONALE

5763 None.

5764 FUTURE DIRECTIONS

5765 None.

5766 SEE ALSO

5767 `csinh()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<complex.h>`

5768 CHANGE HISTORY

5769 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5770 **NAME**
5771 casinl — complex arc sine functions

5772 **SYNOPSIS**
5773 #include <complex.h>
5774 long double complex casinl(long double complex z);

5775 **DESCRIPTION**
5776 Refer to *casin()*.

5777 NAME

5778 catan, catanf, catanl — complex arc tangent functions

5779 SYNOPSIS

```
5780 #include <complex.h>
5781 double complex catan(double complex z);
5782 float complex catanf(float complex z);
5783 long double complex catanl(long double complex z);
```

5784 DESCRIPTION

5785 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5786 conflict between the requirements described here and the ISO C standard is unintentional. This
5787 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5788 These functions shall compute the complex arc tangent of z , with branch cuts outside the
5789 interval $[-i, +i]$ along the imaginary axis.

5790 RETURN VALUE

5791 These functions shall return the complex arc tangent value, in the range of a strip
5792 mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the
5793 real axis.

5794 ERRORS

5795 No errors are defined.

5796 EXAMPLES

5797 None.

5798 APPLICATION USAGE

5799 None.

5800 RATIONALE

5801 None.

5802 FUTURE DIRECTIONS

5803 None.

5804 SEE ALSO

5805 *ctan()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

5806 CHANGE HISTORY

5807 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5808 NAME

5809 catanh, catanhf, catanhl — complex arc hyperbolic tangent functions

5810 SYNOPSIS

```
5811     #include <complex.h>
5812
5813     double complex catanh(double complex z);
5814     float complex catanhf(float complex z);
5815     long double complex catanhl(long double complex z);
```

5815 DESCRIPTION

5816 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5817 conflict between the requirements described here and the ISO C standard is unintentional. This
5818 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5819 These functions shall compute the complex arc hyperbolic tangent of z , with branch cuts outside
5820 the interval $[-1, +1]$ along the real axis.

5821 RETURN VALUE

5822 These functions shall return the complex arc hyperbolic tangent value, in the range of a strip
5823 mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the
5824 imaginary axis.

5825 ERRORS

5826 No errors are defined.

5827 EXAMPLES

5828 None.

5829 APPLICATION USAGE

5830 None.

5831 RATIONALE

5832 None.

5833 FUTURE DIRECTIONS

5834 None.

5835 SEE ALSO

5836 *ctanh()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

5837 CHANGE HISTORY

5838 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

5839 **NAME**
5840 catanl — complex arc tangent functions

5841 **SYNOPSIS**
5842 #include <complex.h>
5843 long double complex catanl(long double complex z);

5844 **DESCRIPTION**
5845 Refer to *catan()*.

5846 NAME

5847 catclose — close a message catalog descriptor

5848 SYNOPSIS

5849 XSI #include <nl_types.h>

5850 int catclose(nl_catd catd);

5851

5852 DESCRIPTION

5853 The *catclose()* function shall close the message catalog identified by *catd*. If a file descriptor is
5854 used to implement the type **nl_catd**, that file descriptor shall be closed.

5855 RETURN VALUE

5856 Upon successful completion, *catclose()* shall return 0; otherwise, -1 shall be returned, and *errno*
5857 set to indicate the error.

5858 ERRORS

5859 The *catclose()* function may fail if:

5860 [EBADF] The catalog descriptor is not valid.

5861 [EINTR] The *catclose()* function was interrupted by a signal.

5862 EXAMPLES

5863 None.

5864 APPLICATION USAGE

5865 None.

5866 RATIONALE

5867 None.

5868 FUTURE DIRECTIONS

5869 None.

5870 SEE ALSO

5871 *catgets()*, *catopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, <nl_types.h>

5872 CHANGE HISTORY

5873 First released in Issue 2.

5874 NAME

5875 catgets — read a program message

5876 SYNOPSIS

5877 XSI #include <nl_types.h>

```
5878        char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);
```

5879

5880 DESCRIPTION

5881 The *catgets()* function shall attempt to read message *msg_id*, in set *set_id*, from the message
5882 catalog identified by *catd*. The *catd* argument is a message catalog descriptor returned from an
5883 earlier call to *catopen()*. The *s* argument points to a default message string which shall be
5884 returned by *catgets()* if it cannot retrieve the identified message.

5885 The *catgets()* function need not be reentrant. A function that is not required to be reentrant is not
5886 required to be thread-safe.

5887 RETURN VALUE

5888 If the identified message is retrieved successfully, *catgets()* shall return a pointer to an internal
5889 buffer area containing the null-terminated message string. If the call is unsuccessful for any
5890 reason, *s* shall be returned and *errno* may be set to indicate the error.

5891 ERRORS

5892 The *catgets()* function may fail if:

- | | |
|-----------------------|--|
| 5893 [EBADF] | The <i>catd</i> argument is not a valid message catalog descriptor open for reading. |
| 5894 [EBADMSG] | The message identified by <i>set_id</i> and <i>msg_id</i> in the specified message catalog
5895 did not satisfy implementation-defined security criteria. |
| 5896 [EINTR] | The read operation was terminated due to the receipt of a signal, and no data
5897 was transferred. |
| 5898 [EINVAL] | The message catalog identified by <i>catd</i> is corrupted. |
| 5899 [ENOMSG] | The message identified by <i>set_id</i> and <i>msg_id</i> is not in the message catalog. |

5900 EXAMPLES

5901 None.

5902 APPLICATION USAGE

5903 None.

5904 RATIONALE

5905 None.

5906 FUTURE DIRECTIONS

5907 None.

5908 SEE ALSO

5909 *catclose()*, *catopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, <nl_types.h>

5910 CHANGE HISTORY

5911 First released in Issue 2.

5912 Issue 5

5913 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

5914 **Issue 6**

5915

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

5916 **NAME**

5917 catopen — open a message catalog

5918 **SYNOPSIS**

5919 XSI #include <nl_types.h>

5920 nl_catd catopen(const char *name, int oflag);

5921

5922 **DESCRIPTION**

5923 The *catopen()* function shall open a message catalog and return a message catalog descriptor.
5924 The *name* argument specifies the name of the message catalog to be opened. If *name* contains a
5925 ' / ', then *name* specifies a complete name for the message catalog. Otherwise, the environment
5926 variable *NLSPATH* is used with *name* substituted for the %N conversion specification (see the
5927 Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables). If
5928 *NLSPATH* exists in the environment when the process starts, then if the process has appropriate
5929 privileges, the behavior of *catopen()* is undefined. If *NLSPATH* does not exist in the environment,
5930 or if a message catalog cannot be found in any of the components specified by *NLSPATH*, then
5931 an implementation-defined default path shall be used. This default may be affected by the
5932 setting of *LC_MESSAGES* if the value of *oflag* is *NL_CAT_LOCALE*, or the *LANG* environment
5933 variable if *oflag* is 0.

5934 A message catalog descriptor shall remain valid in a process until that process closes it, or a
5935 successful call to one of the *exec* functions. A change in the setting of the *LC_MESSAGES*
5936 category may invalidate existing open catalogs.

5937 If a file descriptor is used to implement message catalog descriptors, the *FD_CLOEXEC* flag
5938 shall be set; see <*fcntl.h*>.

5939 If the value of the *oflag* argument is 0, the *LANG* environment variable is used to locate the
5940 catalog without regard to the *LC_MESSAGES* category. If the *oflag* argument is
5941 *NL_CAT_LOCALE*, the *LC_MESSAGES* category is used to locate the message catalog (see the
5942 Base Definitions volume of IEEE Std 1003.1-2001, Section 8.2, Internationalization Variables).

5943 **RETURN VALUE**

5944 Upon successful completion, *catopen()* shall return a message catalog descriptor for use on
5945 subsequent calls to *catgets()* and *catclose()*. Otherwise, *catopen()* shall return (*nl_catd*) -1 and set
5946 *errno* to indicate the error.

5947 **ERRORS**

5948 The *catopen()* function may fail if:

5949 [EACCES] Search permission is denied for the component of the path prefix of the
5950 message catalog or read permission is denied for the message catalog.

5951 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

5952 [ENAMETOOLONG] The length of a pathname of the message catalog exceeds {PATH_MAX} or a
5953 pathname component is longer than {NAME_MAX}.

5955 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
5956 whose length exceeds {PATH_MAX}.

5958 [ENFILE] Too many files are currently open in the system.

5959 [ENOENT] The message catalog does not exist or the *name* argument points to an empty
5960 string.

- 5961 [ENOMEM] Insufficient storage space is available.
5962 [ENOTDIR] A component of the path prefix of the message catalog is not a directory.

5963 **EXAMPLES**

5964 None.

5965 **APPLICATION USAGE**

5966 Some implementations of *catopen()* use *malloc()* to allocate space for internal buffer areas. The
5967 *catopen()* function may fail if there is insufficient storage space available to accommodate these
5968 buffers.

5969 Conforming applications must assume that message catalog descriptors are not valid after a call
5970 to one of the *exec* functions.

5971 Application writers should be aware that guidelines for the location of message catalogs have
5972 not yet been developed. Therefore they should take care to avoid conflicting with catalogs used
5973 by other applications and the standard utilities.

5974 **RATIONALE**

5975 None.

5976 **FUTURE DIRECTIONS**

5977 None.

5978 **SEE ALSO**

5979 *catclose()*, *catgets()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>,
5980 <nl_types.h>, the Shell and Utilities volume of IEEE Std 1003.1-2001

5981 **CHANGE HISTORY**

5982 First released in Issue 2.

5983 NAME

5984 cbrt, cbrtf, cbtrt — cube root functions

5985 SYNOPSIS

```
5986     #include <math.h>
5987
5988     double cbrt(double x);
5989     float cbrtf(float x);
5990     long double cbtrt(long double x);
```

5990 DESCRIPTION

5991 CX The functionality described on this reference page is aligned with the ISO C standard. Any
5992 conflict between the requirements described here and the ISO C standard is unintentional. This
5993 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

5994 These functions shall compute the real cube root of their argument *x*.

5995 RETURN VALUE

5996 Upon successful completion, these functions shall return the cube root of *x*.

5997 MX If *x* is NaN, a NaN shall be returned.

5998 If *x* is ±0 or ±Inf, *x* shall be returned.

5999 ERRORS

6000 No errors are defined.

6001 EXAMPLES

6002 None.

6003 APPLICATION USAGE

6004 None.

6005 RATIONALE

6006 For some applications, a true cube root function, which returns negative results for negative
6007 arguments, is more appropriate than *pow(x, 1.0/3.0)*, which returns a NaN for *x* less than 0.

6008 FUTURE DIRECTIONS

6009 None.

6010 SEE ALSO

6011 The Base Definitions volume of IEEE Std 1003.1-2001, <**math.h**>

6012 CHANGE HISTORY

6013 First released in Issue 4, Version 2.

6014 Issue 5

6015 Moved from X/OPEN UNIX extension to BASE.

6016 Issue 6

6017 The *cbrt()* function is no longer marked as an extension.

6018 The *cbrtf()* and *cbtrt()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

6019 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
6020 revised to align with the ISO/IEC 9899:1999 standard.

6021 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
6022 marked.

6023 NAME

6024 `ccos, ccosf, ccosl` — complex cosine functions

6025 SYNOPSIS

```
6026       #include <complex.h>
6027       double complex ccos(double complex z);
6028       float complex ccosf(float complex z);
6029       long double complex ccosl(long double complex z);
```

6030 DESCRIPTION

6031 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6032 conflict between the requirements described here and the ISO C standard is unintentional. This
6033 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6034 These functions shall compute the complex cosine of *z*.

6035 RETURN VALUE

6036 These functions shall return the complex cosine value.

6037 ERRORS

6038 No errors are defined.

6039 EXAMPLES

6040 None.

6041 APPLICATION USAGE

6042 None.

6043 RATIONALE

6044 None.

6045 FUTURE DIRECTIONS

6046 None.

6047 SEE ALSO

6048 `cacos()`, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

6049 CHANGE HISTORY

6050 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6051 NAME

6052 ccosh, ccoshf, ccoshl — complex hyperbolic cosine functions

6053 SYNOPSIS

```
6054     #include <complex.h>
6055
6056     double complex ccosh(double complex z);
6057     float complex ccoshf(float complex z);
6058     long double complex ccoshl(long double complex z);
```

6058 DESCRIPTION

6059 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6060 conflict between the requirements described here and the ISO C standard is unintentional. This
6061 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6062 These functions shall compute the complex hyperbolic cosine of *z*.

6063 RETURN VALUE

6064 These functions shall return the complex hyperbolic cosine value.

6065 ERRORS

6066 No errors are defined.

6067 EXAMPLES

6068 None.

6069 APPLICATION USAGE

6070 None.

6071 RATIONALE

6072 None.

6073 FUTURE DIRECTIONS

6074 None.

6075 SEE ALSO

6076 cacosh(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

6077 CHANGE HISTORY

6078 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6079 **NAME**
6080 ccosl — complex cosine functions

6081 **SYNOPSIS**
6082 #include <complex.h>
6083 long double complex ccosl(long double complex z);

6084 **DESCRIPTION**
6085 Refer to *ccos()*.

6086 NAME

6087 ceil, ceilf, ceill — ceiling value function

6088 SYNOPSIS

```
6089 #include <math.h>
6090
6091     double ceil(double x);
6092     float ceilf(float x);
6093     long double ceill(long double x);
```

6093 DESCRIPTION

6094 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6095 conflict between the requirements described here and the ISO C standard is unintentional. This
6096 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6097 These functions shall compute the smallest integral value not less than *x*.

6098 An application wishing to check for error situations should set *errno* to zero and call
6099 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
6100 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
6101 zero, an error has occurred.

6102 RETURN VALUE

6103 Upon successful completion, *ceil()*, *ceilf()*, and *ceill()* shall return the smallest integral value not
6104 less than *x*, expressed as a type **double**, **float**, or **long double**, respectively.

6105 MX If *x* is NaN, a NaN shall be returned.

6106 If *x* is ±0 or ±Inf, *x* shall be returned.

6107 XSI If the correct value would cause overflow, a range error shall occur and *ceil()*, *ceilf()*, and *ceill()*
6108 shall return the value of the macro **HUGE_VAL**, **HUGE_VALF**, and **HUGE_VALL**, respectively.

6109 ERRORS

6110 These functions shall fail if:

6111 XSI Range Error The result overflows.

6112 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
6113 then *errno* shall be set to [ERANGE]. If the integer expression
6114 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
6115 floating-point exception shall be raised.

6116 EXAMPLES

6117 None.

6118 APPLICATION USAGE

6119 The integral value returned by these functions need not be expressible as an **int** or **long**. The
6120 return value should be tested before assigning it to an integer type to avoid the undefined results
6121 of an integer overflow.

6122 The *ceil()* function can only overflow when the floating-point representation has
6123 **DBL_MANT_DIG** > **DBL_MAX_EXP**.

6124 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
6125 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

6126 RATIONALE

6127 None.

6128 FUTURE DIRECTIONS

6129 None.

6130 SEE ALSO

6131 *feclearexcept()*, *fetestexcept()*, *floor()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
6132 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

6133 CHANGE HISTORY

6134 First released in Issue 1. Derived from Issue 1 of the SVID.

6135 Issue 5

6136 The DESCRIPTION is updated to indicate how an application should check for an error. This
6137 text was previously published in the APPLICATION USAGE section.

6138 Issue 6

6139 The *ceilf()* and *ceil()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

6140 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
6141 revised to align with the ISO/IEC 9899:1999 standard.

6142 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
6143 marked.

6144 NAME

6145 `cexp, cexpf, cexpl — complex exponential functions`

6146 SYNOPSIS

```
6147        #include <complex.h>
6148        double complex cexp(double complex z);
6149        float complex cexpf(float complex z);
6150        long double complex cexpl(long double complex z);
```

6151 DESCRIPTION

6152 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6153 conflict between the requirements described here and the ISO C standard is unintentional. This
6154 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6155 These functions shall compute the complex exponent of z , defined as e^z .

6156 RETURN VALUE

6157 These functions shall return the complex exponential value of z .

6158 ERRORS

6159 No errors are defined.

6160 EXAMPLES

6161 None.

6162 APPLICATION USAGE

6163 None.

6164 RATIONALE

6165 None.

6166 FUTURE DIRECTIONS

6167 None.

6168 SEE ALSO

6169 `clog()`, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

6170 CHANGE HISTORY

6171 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6172 NAME

6173 *cfgetispeed* — get input baud rate

6174 SYNOPSIS

```
6175        #include <termios.h>
6176
6176        speed_t cfgetispeed(const struct termios *termios_p);
```

6177 DESCRIPTION

6178 The *cfgetispeed()* function shall extract the input baud rate from the **termios** structure to which
6179 the *termios_p* argument points.

6180 This function shall return exactly the value in the **termios** data structure, without interpretation.

6181 RETURN VALUE

6182 Upon successful completion, *cfgetispeed()* shall return a value of type **speed_t** representing the
6183 input baud rate.

6184 ERRORS

6185 No errors are defined.

6186 EXAMPLES

6187 None.

6188 APPLICATION USAGE

6189 None.

6190 RATIONALE

6191 The term “baud” is used historically here, but is not technically correct. This is properly “bits
6192 per second”, which may not be the same as baud. However, the term is used because of the
6193 historical usage and understanding.

6194 The *cfgetospeed()*, *cfgetispeed()*, *cfsetospeed()*, and *cfsetispeed()* functions do not take arguments as
6195 numbers, but rather as symbolic names. There are two reasons for this:

- 6196 1. Historically, numbers were not used because of the way the rate was stored in the data
6197 structure. This is retained even though a function is now used.
- 6198 2. More importantly, only a limited set of possible rates is at all portable, and this constrains
6199 the application to that set.

6200 There is nothing to prevent an implementation accepting as an extension a number (such as 126),
6201 and since the encoding of the Bxxx symbols is not specified, this can be done to avoid
6202 introducing ambiguity.

6203 Setting the input baud rate to zero was a mechanism to allow for split baud rates. Clarifications
6204 in this volume of IEEE Std 1003.1-2001 have made it possible to determine whether split rates are
6205 supported and to support them without having to treat zero as a special case. Since this
6206 functionality is also confusing, it has been declared obsolescent. The 0 argument referred to is
6207 the literal constant 0, not the symbolic constant B0. This volume of IEEE Std 1003.1-2001 does not
6208 preclude B0 from being defined as the value 0; in fact, implementations would likely benefit
6209 from the two being equivalent. This volume of IEEE Std 1003.1-2001 does not fully specify
6210 whether the previous *cfsetispeed()* value is retained after a *tcgetattr()* as the actual value or as
6211 zero. Therefore, conforming applications should always set both the input speed and output
6212 speed when setting either.

6213 In historical implementations, the baud rate information is traditionally kept in **c_cflag**.
6214 Applications should be written to presume that this might be the case (and thus not blindly copy
6215 **c_cflag**), but not to rely on it in case it is in some other field of the structure. Setting the **c_cflag**
6216 field absolutely after setting a baud rate is a non-portable action because of this. In general, the

6217 unused parts of the flag fields might be used by the implementation and should not be blindly
6218 copied from the descriptions of one terminal device to another.

6219 **FUTURE DIRECTIONS**

6220 None.

6221 **SEE ALSO**

6222 *cfgetospeed()*, *cfsetispeed()*, *cfsetospeed()*, *tcgetattr()*, the Base Definitions volume of
6223 IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <**termios.h**>

6224 **CHANGE HISTORY**

6225 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

6226 **NAME**
6227 *cfgetospeed* — get output baud rate

6228 **SYNOPSIS**
6229 #include <termios.h>
6230
6230 speed_t cfgetospeed(const struct termios **termios_p*);

6231 **DESCRIPTION**
6232 The *cfgetospeed()* function shall extract the output baud rate from the **termios** structure to which
6233 the *termios_p* argument points.

6234 This function shall return exactly the value in the **termios** data structure, without interpretation.

6235 **RETURN VALUE**
6236 Upon successful completion, *cfgetospeed()* shall return a value of type **speed_t** representing the
6237 output baud rate.

6238 **ERRORS**
6239 No errors are defined.

6240 **EXAMPLES**
6241 None.

6242 **APPLICATION USAGE**
6243 None.

6244 **RATIONALE**
6245 Refer to *cfgetispeed()*.

6246 **FUTURE DIRECTIONS**
6247 None.

6248 **SEE ALSO**
6249 *cfgetispeed()*, *cfsetispeed()*, *cfsetospeed()*, *tcgetattr()*, the Base Definitions volume of
6250 IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>

6251 **CHANGE HISTORY**
6252 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

6253 NAME

6254 *cfsetispeed* — set input baud rate

6255 SYNOPSIS

```
6256        #include <termios.h>
6257
6258        int cfsetispeed(struct termios *termios_p, speed_t speed);
```

6258 DESCRIPTION

6259 The *cfsetispeed()* function shall set the input baud rate stored in the structure pointed to by *termios_p* to *speed*.

6261 There shall be no effect on the baud rates set in the hardware until a subsequent successful call
6262 to *tcsetattr()* with the same **termios** structure. Similarly, errors resulting from attempts to set
6263 baud rates not supported by the terminal device need not be detected until the *tcsetattr()*
6264 function is called.

6265 RETURN VALUE

6266 Upon successful completion, *cfsetispeed()* shall return 0; otherwise, -1 shall be returned, and
6267 *errno* may be set to indicate the error.

6268 ERRORS

6269 The *cfsetispeed()* function may fail if:

6270 [EINVAL] The *speed* value is not a valid baud rate.

6271 [EINVAL] The value of *speed* is outside the range of possible speed values as specified in
6272 <termios.h>.

6273 EXAMPLES

6274 None.

6275 APPLICATION USAGE

6276 None.

6277 RATIONALE

6278 Refer to *cfgetispeed()*.

6279 FUTURE DIRECTIONS

6280 None.

6281 SEE ALSO

6282 *cfgetispeed()*, *cfgetospeed()*, *cfsetospeed()*, *tcsetattr()*, the Base Definitions volume of
6283 IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>

6284 CHANGE HISTORY

6285 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

6286 Issue 6

6287 The following new requirements on POSIX implementations derive from alignment with the
6288 Single UNIX Specification:

- The optional setting of *errno* and the [EINVAL] error conditions are added.

6290 NAME

6291 *cfsetospeed* — set output baud rate

6292 SYNOPSIS

```
6293        #include <termios.h>
6294        int cfsetospeed(struct termios *termios_p, speed_t speed);
```

6295 DESCRIPTION

6296 The *cfsetospeed()* function shall set the output baud rate stored in the structure pointed to by *termios_p* to *speed*.

6298 There shall be no effect on the baud rates set in the hardware until a subsequent successful call
6299 to *tcsetattr()* with the same **termios** structure. Similarly, errors resulting from attempts to set
6300 baud rates not supported by the terminal device need not be detected until the *tcsetattr()*
6301 function is called.

6302 RETURN VALUE

6303 Upon successful completion, *cfsetospeed()* shall return 0; otherwise, it shall return -1 and *errno*
6304 may be set to indicate the error.

6305 ERRORS

6306 The *cfsetospeed()* function may fail if:

6307 [EINVAL] The *speed* value is not a valid baud rate.

6308 [EINVAL] The value of *speed* is outside the range of possible speed values as specified in
6309 <termios.h>.

6310 EXAMPLES

6311 None.

6312 APPLICATION USAGE

6313 None.

6314 RATIONALE

6315 Refer to *cfgetispeed()*.

6316 FUTURE DIRECTIONS

6317 None.

6318 SEE ALSO

6319 *cfgetispeed()*, *cfgetospeed()*, *cfsetispeed()*, *tcsetattr()*, the Base Definitions volume of
6320 IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>

6321 CHANGE HISTORY

6322 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

6323 Issue 6

6324 The following new requirements on POSIX implementations derive from alignment with the
6325 Single UNIX Specification:

- The optional setting of *errno* and the [EINVAL] error conditions are added.

6327 NAME

6328 chdir — change working directory

6329 SYNOPSIS

```
6330        #include <unistd.h>
6331        int chdir(const char *path);
```

6332 DESCRIPTION

6333 The *chdir()* function shall cause the directory named by the pathname pointed to by the *path* argument to become the current working directory; that is, the starting point for path searches for pathnames not beginning with '/'.

6336 RETURN VALUE

6337 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned, the current
6338 working directory shall remain unchanged, and *errno* shall be set to indicate the error.

6339 ERRORS

6340 The *chdir()* function shall fail if:

6341 [EACCES] Search permission is denied for any component of the pathname.

6342 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
6343 argument.

6344 [ENAMETOOLONG]

6345 The length of the *path* argument exceeds {PATH_MAX} or a pathname
6346 component is longer than {NAME_MAX}.

6347 [ENOENT] A component of *path* does not name an existing directory or *path* is an empty
6348 string.

6349 [ENOTDIR] A component of the pathname is not a directory.

6350 The *chdir()* function may fail if:

6351 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
6352 resolution of the *path* argument.

6353 [ENAMETOOLONG]

6354 As a result of encountering a symbolic link in resolution of the *path* argument,
6355 the length of the substituted pathname string exceeded {PATH_MAX}.

6356 EXAMPLES**6357 Changing the Current Working Directory**

6358 The following example makes the value pointed to by **directory**, /tmp, the current working
6359 directory.

```
6360        #include <unistd.h>
6361        ...
6362        char *directory = "/tmp";
6363        int ret;
6364
6365        ret = chdir (directory);
```

6365 **APPLICATION USAGE**

6366 None.

6367 **RATIONALE**

6368 The *chdir()* function only affects the working directory of the current process.

6369 **FUTURE DIRECTIONS**

6370 None.

6371 **SEE ALSO**

6372 *getcwd()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

6373 **CHANGE HISTORY**

6374 First released in Issue 1. Derived from Issue 1 of the SVID.

6375 **Issue 6**

6376 The APPLICATION USAGE section is added.

6377 The following new requirements on POSIX implementations derive from alignment with the
6378 Single UNIX Specification:

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

6381 The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

6383 **NAME**

6384 chmod — change mode of a file

6385 **SYNOPSIS**

```
6386        #include <sys/stat.h>
6387        int chmod(const char *path, mode_t mode);
```

6388 **DESCRIPTION**

6389 XSI The *chmod()* function shall change S_ISUID, S_ISGID, S_ISVTX, and the file permission bits of the file named by the pathname pointed to by the *path* argument to the corresponding bits in the *mode* argument. The application shall ensure that the effective user ID of the process matches the owner of the file or the process has appropriate privileges in order to do this.

6393 XSI S_ISUID, S_ISGID, S_ISVTX, and the file permission bits are described in <sys/stat.h>.

6394 If the calling process does not have appropriate privileges, and if the group ID of the file does not match the effective group ID or one of the supplementary group IDs and if the file is a regular file, bit S_ISGID (set-group-ID on execution) in the file's mode shall be cleared upon successful return from *chmod()*.

6398 Additional implementation-defined restrictions may cause the S_ISUID and S_ISGID bits in *mode* to be ignored.

6400 The effect on file descriptors for files open at the time of a call to *chmod()* is implementation-defined.

6402 Upon successful completion, *chmod()* shall mark for update the *st_ctime* field of the file.

6403 **RETURN VALUE**

6404 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to indicate the error. If -1 is returned, no change to the file mode occurs.

6406 **ERRORS**

6407 The *chmod()* function shall fail if:

6408 [EACCES] Search permission is denied on a component of the path prefix.

6409 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

6411 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

6414 [ENOTDIR] A component of the path prefix is not a directory.

6415 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

6416 [EPERM] The effective user ID does not match the owner of the file and the process does not have appropriate privileges.

6418 [EROFS] The named file resides on a read-only file system.

6419 The *chmod()* function may fail if:

6420 [EINTR] A signal was caught during execution of the function.

6421 [EINVAL] The value of the *mode* argument is invalid.

6422 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

6424 [ENAMETOOLONG]
6425 As a result of encountering a symbolic link in resolution of the *path* argument,
6426 the length of the substituted pathname strings exceeded {PATH_MAX}.

6427 EXAMPLES

6428 Setting Read Permissions for User, Group, and Others

6429 The following example sets read permissions for the owner, group, and others.

```
6430 #include <sys/stat.h>
6431 const char *path;
6432 ...
6433 chmod(path, S_IRUSR|S_IRGRP|S_IROTH);
```

6434 Setting Read, Write, and Execute Permissions for the Owner Only

6435 The following example sets read, write, and execute permissions for the owner, and no
6436 permissions for group and others.

```
6437 #include <sys/stat.h>
6438 const char *path;
6439 ...
6440 chmod(path, S_IRWXU);
```

6441 Setting Different Permissions for Owner, Group, and Other

6442 The following example sets owner permissions for CHANGETFILE to read, write, and execute,
6443 group permissions to read and execute, and other permissions to read.

```
6444 #include <sys/stat.h>
6445 #define CHANGETFILE "/etc/myfile"
6446 ...
6447 chmod(CHANGETFILE, S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH);
```

6448 Setting and Checking File Permissions

6449 The following example sets the file permission bits for a file named /home/cnd/mod1, then calls
6450 the *stat()* function to verify the permissions.

```
6451 #include <sys/types.h>
6452 #include <sys/stat.h>
6453 int status;
6454 struct stat buffer
6455 ...
6456 chmod("home/cnd/mod1", S_IRWXU|S_IRWXG|S_IROTH|S_IWOTH);
6457 status = stat("home/cnd/mod1", &buffer);
```

6458 APPLICATION USAGE

6459 In order to ensure that the S_ISUID and S_ISGID bits are set, an application requiring this should
6460 use *stat()* after a successful *chmod()* to verify this.

6461 Any file descriptors currently open by any process on the file could possibly become invalid if
6462 the mode of the file is changed to a value which would deny access to that process. One

6463 situation where this could occur is on a stateless file system. This behavior will not occur in a
6464 conforming environment.

6465 RATIONALE

6466 This volume of IEEE Std 1003.1-2001 specifies that the S_ISGID bit is cleared by *chmod()* on a
6467 regular file under certain conditions. This is specified on the assumption that regular files may
6468 be executed, and the system should prevent users from making executable *setgid()* files perform
6469 with privileges that the caller does not have. On implementations that support execution of
6470 other file types, the S_ISGID bit should be cleared for those file types under the same
6471 circumstances.

6472 Implementations that use the S_ISUID bit to indicate some other function (for example,
6473 mandatory record locking) on non-executable files need not clear this bit on writing. They
6474 should clear the bit for executable files and any other cases where the bit grants special powers
6475 to processes that change the file contents. Similar comments apply to the S_ISGID bit.

6476 FUTURE DIRECTIONS

6477 None.

6478 SEE ALSO

6479 *chown()*, *mkdir()*, *mkfifo()*, *open()*, *stat()*, *statvfs()*, the Base Definitions volume of
6480 IEEE Std 1003.1-2001, <sys/stat.h>, <sys/types.h>

6481 CHANGE HISTORY

6482 First released in Issue 1. Derived from Issue 1 of the SVID.

6483 Issue 6

6484 The following new requirements on POSIX implementations derive from alignment with the
6485 Single UNIX Specification:

- 6486 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
6487 required for conforming implementations of previous POSIX specifications, it was not
6488 required for UNIX applications.
- 6489 • The [EINVAL] and [EINTR] optional error conditions are added.
- 6490 • A second [ENAMETOOLONG] is added as an optional error condition.

6491 The following changes were made to align with the IEEE P1003.1a draft standard:

- 6492 • The [ELOOP] optional error condition is added.

6493 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

6494 **NAME**

6495 chown — change owner and group of a file

6496 **SYNOPSIS**

6497

```
#include <unistd.h>
```


6498

```
int chown(const char *path, uid_t owner, gid_t group);
```

6499 **DESCRIPTION**

6500 The *chown()* function shall change the user and group ownership of a file.

6501 The *path* argument points to a pathname naming a file. The user ID and group ID of the named
6502 file shall be set to the numeric values contained in *owner* and *group*, respectively.

6503 Only processes with an effective user ID equal to the user ID of the file or with appropriate
6504 privileges may change the ownership of a file. If *_POSIX_CHOWN_RESTRICTED* is in effect for
6505 *path*:

- 6506 • Changing the user ID is restricted to processes with appropriate privileges.
- 6507 • Changing the group ID is permitted to a process with an effective user ID equal to the user
6508 ID of the file, but without appropriate privileges, if and only if *owner* is equal to the file's user
6509 ID or (*uid_t*)–1 and *group* is equal either to the calling process' effective group ID or to one of
6510 its supplementary group IDs.

6511 If the specified file is a regular file, one or more of the S_IXUSR, S_IXGRP, or S_IXOTH bits of
6512 the file mode are set, and the process does not have appropriate privileges, the set-user-ID
6513 (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode shall be cleared upon successful
6514 return from *chown()*. If the specified file is a regular file, one or more of the S_IXUSR, S_IXGRP,
6515 or S_IXOTH bits of the file mode are set, and the process has appropriate privileges, it is
6516 implementation-defined whether the set-user-ID and set-group-ID bits are altered. If the *chown()*
6517 function is successfully invoked on a file that is not a regular file and one or more of the
6518 S_IXUSR, S_IXGRP, or S_IXOTH bits of the file mode are set, the set-user-ID and set-group-ID
6519 bits may be cleared.

6520 If *owner* or *group* is specified as (*uid_t*)–1 or (*gid_t*)–1, respectively, the corresponding ID of the
6521 file shall not be changed. If both *owner* and *group* are –1, the times need not be updated.

6522 Upon successful completion, *chown()* shall mark for update the *st_ctime* field of the file.

6523 **RETURN VALUE**

6524 Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to
6525 indicate the error. If –1 is returned, no changes are made in the user ID and group ID of the file.

6526 **ERRORS**

6527 The *chown()* function shall fail if:

- | | |
|---------------------|--|
| 6528 [EACCES] | Search permission is denied on a component of the path prefix. |
| 6529 [ELOOP] | A loop exists in symbolic links encountered during resolution of the <i>path</i> argument. |
| 6531 [ENAMETOOLONG] | The length of the <i>path</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}. |
| 6534 [ENOTDIR] | A component of the path prefix is not a directory. |
| 6535 [ENOENT] | A component of <i>path</i> does not name an existing file or <i>path</i> is an empty string. |

6536	[EPERM]	The effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges and _POSIX_CHOWN_RESTRICTED indicates that such privilege is required.
6537		
6538		
6539	[EROFS]	The named file resides on a read-only file system.
6540		The <i>chown()</i> function may fail if:
6541	[EIO]	An I/O error occurred while reading or writing to the file system.
6542	[EINTR]	The <i>chown()</i> function was interrupted by a signal which was caught.
6543	[EINVAL]	The owner or group ID supplied is not a value supported by the implementation.
6544		
6545	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path</i> argument.
6546		
6547	[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}.
6548		
6549		

6550 EXAMPLES

6551 None.

6552 APPLICATION USAGE

6553 Although *chown()* can be used on some implementations by the file owner to change the owner
6554 and group to any desired values, the only portable use of this function is to change the group of
6555 a file to the effective GID of the calling process or to a member of its group set.

6556 RATIONALE

6557 System III and System V allow a user to give away files; that is, the owner of a file may change
6558 its user ID to anything. This is a serious problem for implementations that are intended to meet
6559 government security regulations. Version 7 and 4.3 BSD permit only the superuser to change the
6560 user ID of a file. Some government agencies (usually not ones concerned directly with security)
6561 find this limitation too confining. This volume of IEEE Std 1003.1-2001 uses *may* to permit secure
6562 implementations while not disallowing System V.

6563 System III and System V allow the owner of a file to change the group ID to anything. Version 7
6564 permits only the superuser to change the group ID of a file. 4.3 BSD permits the owner to
6565 change the group ID of a file to its effective group ID or to any of the groups in the list of
6566 supplementary group IDs, but to no others.

6567 The POSIX.1-1990 standard requires that the *chown()* function invoked by a non-appropriate
6568 privileged process clear the S_ISGID and the S_ISUID bits for regular files, and permits them to
6569 be cleared for other types of files. This is so that changes in accessibility do not accidentally
6570 cause files to become security holes. Unfortunately, requiring these bits to be cleared on non-
6571 executable data files also clears the mandatory file locking bit (shared with S_ISGID), which is
6572 an extension on many implementations (it first appeared in System V). These bits should only be
6573 required to be cleared on regular files that have one or more of their execute bits set.

6574 FUTURE DIRECTIONS

6575 None.

6576 SEE ALSO

6577 *chmod()*, *pathconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>,
6578 <unistd.h>

6579 **CHANGE HISTORY**

6580 First released in Issue 1. Derived from Issue 1 of the SVID.

6581 **Issue 6**

6582 The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- 6583 • The wording describing the optional dependency on _POSIX_CHOWN_RESTRICTED is
6584 restored.
- 6585 • The [EPERM] error is restored as an error dependent on _POSIX_CHOWN_RESTRICTED.
6586 This is since its operand is a pathname and applications should be aware that the error may
6587 not occur for that pathname if the file system does not support
6588 _POSIX_CHOWN_RESTRICTED.

6589 The following new requirements on POSIX implementations derive from alignment with the
6590 Single UNIX Specification:

- 6591 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
6592 required for conforming implementations of previous POSIX specifications, it was not
6593 required for UNIX applications.
- 6594 • The value for *owner* of (uid_t)-1 allows the use of -1 by the owner of a file to change the
6595 group ID only. A corresponding change is made for group.
- 6596 • The [ELOOP] mandatory error condition is added.
- 6597 • The [EIO] and [EINTR] optional error conditions are added.
- 6598 • A second [ENAMETOOLONG] is added as an optional error condition.

6599 The following changes were made to align with the IEEE P1003.1a draft standard:

- 6600 • Clarification is added that the S_ISUID and S_ISGID bits do not need to be cleared when the
6601 process has appropriate privileges.
- 6602 • The [ELOOP] optional error condition is added.

6603 NAME

6604 `cimag`, `cimagf`, `cimagl` — complex imaginary functions

6605 SYNOPSIS

```
6606        #include <complex.h>
6607
6608        double cimag(double complex z);
6609        float cimagf(float complex z);
6610        long double cimagl(long double complex z);
```

6610 DESCRIPTION

6611 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6612 conflict between the requirements described here and the ISO C standard is unintentional. This
6613 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6614 These functions shall compute the imaginary part of *z*.

6615 RETURN VALUE

6616 These functions shall return the imaginary part value (as a real).

6617 ERRORS

6618 No errors are defined.

6619 EXAMPLES

6620 None.

6621 APPLICATION USAGE

6622 For a variable *z* of complex type:

```
6623        z == creal(z) + cimag(z)*I
```

6624 RATIONALE

6625 None.

6626 FUTURE DIRECTIONS

6627 None.

6628 SEE ALSO

6629 `carg()`, `conj()`, `cproj()`, `creal()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<complex.h>`

6630 CHANGE HISTORY

6631 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6632 NAME

6633 clearerr — clear indicators on a stream

6634 SYNOPSIS

6635 #include <stdio.h>

6636 void clearerr(FILE *stream);

6637 DESCRIPTION

6638 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6639 conflict between the requirements described here and the ISO C standard is unintentional. This
6640 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6641 The *clearerr()* function shall clear the end-of-file and error indicators for the stream to which
6642 *stream* points.

6643 RETURN VALUE

6644 The *clearerr()* function shall not return a value.

6645 ERRORS

6646 No errors are defined.

6647 EXAMPLES

6648 None.

6649 APPLICATION USAGE

6650 None.

6651 RATIONALE

6652 None.

6653 FUTURE DIRECTIONS

6654 None.

6655 SEE ALSO

6656 The Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

6657 CHANGE HISTORY

6658 First released in Issue 1. Derived from Issue 1 of the SVID.

6659 **NAME**

6660 *clock* — report CPU time used

6661 **SYNOPSIS**

```
6662     #include <time.h>
6663
6664     clock_t clock(void);
```

6664 **DESCRIPTION**

6665 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6666 conflict between the requirements described here and the ISO C standard is unintentional. This
6667 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6668 The *clock()* function shall return the implementation's best approximation to the processor time
6669 used by the process since the beginning of an implementation-defined era related only to the
6670 process invocation.

6671 **RETURN VALUE**

6672 To determine the time in seconds, the value returned by *clock()* should be divided by the value
6673 XSI of the macro **CLOCKS_PER_SEC**. **CLOCKS_PER_SEC** is defined to be one million in **<time.h>**.
6674 If the processor time used is not available or its value cannot be represented, the function shall
6675 return the value (*clock_t*)–1.

6676 **ERRORS**

6677 No errors are defined.

6678 **EXAMPLES**

6679 None.

6680 **APPLICATION USAGE**

6681 In order to measure the time spent in a program, *clock()* should be called at the start of the
6682 program and its return value subtracted from the value returned by subsequent calls. The value
6683 returned by *clock()* is defined for compatibility across systems that have clocks with different
6684 resolutions. The resolution on any particular system need not be to microsecond accuracy.

6685 The value returned by *clock()* may wrap around on some implementations. For example, on a
6686 machine with 32-bit values for **clock_t**, it wraps after 2 147 seconds or 36 minutes.

6687 **RATIONALE**

6688 None.

6689 **FUTURE DIRECTIONS**

6690 None.

6691 **SEE ALSO**

6692 *asctime()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *mktme()*, *strftime()*, *strptime()*, *time()*, *utime()*,
6693 the Base Definitions volume of IEEE Std 1003.1-2001, **<time.h>**

6694 **CHANGE HISTORY**

6695 First released in Issue 1. Derived from Issue 1 of the SVID.

6696 NAME

6697 clock_getcpuclockid — access a process CPU-time clock (**ADVANCED REALTIME**)

6698 SYNOPSIS

6699 CPT #include <time.h>

```
6700     int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
```

6701

6702 DESCRIPTION

6703 The *clock_getcpuclockid()* function shall return the clock ID of the CPU-time clock of the process
6704 specified by *pid*. If the process described by *pid* exists and the calling process has permission,
6705 the clock ID of this clock shall be returned in *clock_id*.

6706 If *pid* is zero, the *clock_getcpuclockid()* function shall return the clock ID of the CPU-time clock of
6707 the process making the call, in *clock_id*.

6708 The conditions under which one process has permission to obtain the CPU-time clock ID of
6709 other processes are implementation-defined.

6710 RETURN VALUE

6711 Upon successful completion, *clock_getcpuclockid()* shall return zero; otherwise, an error number
6712 shall be returned to indicate the error.

6713 ERRORS

6714 The *clock_getcpuclockid()* function shall fail if:

6715 [EPERM] The requesting process does not have permission to access the CPU-time
6716 clock for the process.

6717 The *clock_getcpuclockid()* function may fail if:

6718 [ESRCH] No process can be found corresponding to the process specified by *pid*.

6719 EXAMPLES

6720 None.

6721 APPLICATION USAGE

6722 The *clock_getcpuclockid()* function is part of the Process CPU-Time Clocks option and need not
6723 be provided on all implementations.

6724 RATIONALE

6725 None.

6726 FUTURE DIRECTIONS

6727 None.

6728 SEE ALSO

6729 *clock_getres()*, *timer_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

6730 CHANGE HISTORY

6731 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

6732 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

6733 NAME

6734 clock_getres, clock_gettime, clock_settime — clock and timer functions (**REALTIME**)

6735 SYNOPSIS

6736 TMR

```
#include <time.h>
```

```
6737     int clock_getres(clockid_t clock_id, struct timespec *res);
6738     int clock_gettime(clockid_t clock_id, struct timespec *tp);
6739     int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

6740

6741 DESCRIPTION

6742 The *clock_getres()* function shall return the resolution of any clock. Clock resolutions are
 6743 implementation-defined and cannot be set by a process. If the argument *res* is not NULL, the
 6744 resolution of the specified clock shall be stored in the location pointed to by *res*. If *res* is NULL,
 6745 the clock resolution is not returned. If the *time* argument of *clock_settime()* is not a multiple of *res*,
 6746 then the value is truncated to a multiple of *res*.

6747 The *clock_gettime()* function shall return the current value *tp* for the specified clock, *clock_id*.

6748 The *clock_settime()* function shall set the specified clock, *clock_id*, to the value specified by *tp*.
 6749 Time values that are between two consecutive non-negative integer multiples of the resolution
 6750 of the specified clock shall be truncated down to the smaller multiple of the resolution.

6751 A clock may be system-wide (that is, visible to all processes) or per-process (measuring time that
 6752 is meaningful only within a process). All implementations shall support a *clock_id* of
 6753 CLOCK_REALTIME as defined in **<time.h>**. This clock represents the realtime clock for the
 6754 system. For this clock, the values returned by *clock_gettime()* and specified by *clock_settime()*
 6755 represent the amount of time (in seconds and nanoseconds) since the Epoch. An implementation
 6756 may also support additional clocks. The interpretation of time values for these clocks is
 6757 unspecified.

6758 If the value of the CLOCK_REALTIME clock is set via *clock_settime()*, the new value of the clock
 6759 shall be used to determine the time of expiration for absolute time services based upon the
 6760 CLOCK_REALTIME clock. This applies to the time at which armed absolute timers expire. If the
 6761 absolute time requested at the invocation of such a time service is before the new value of the
 6762 clock, the time service shall expire immediately as if the clock had reached the requested time
 6763 normally.

6764 Setting the value of the CLOCK_REALTIME clock via *clock_settime()* shall have no effect on
 6765 threads that are blocked waiting for a relative time service based upon this clock, including the
 6766 *nanosleep()* function; nor on the expiration of relative timers based upon this clock.
 6767 Consequently, these time services shall expire when the requested relative interval elapses,
 6768 independently of the new or old value of the clock.

6769 MON If the Monotonic Clock option is supported, all implementations shall support a *clock_id* of
 6770 CLOCK_MONOTONIC defined in **<time.h>**. This clock represents the monotonic clock for the
 6771 system. For this clock, the value returned by *clock_gettime()* represents the amount of time (in
 6772 seconds and nanoseconds) since an unspecified point in the past (for example, system start-up
 6773 time, or the Epoch). This point does not change after system start-up time. The value of the
 6774 CLOCK_MONOTONIC clock cannot be set via *clock_settime()*. This function shall fail if it is
 6775 invoked with a *clock_id* argument of CLOCK_MONOTONIC.

6776 The effect of setting a clock via *clock_settime()* on armed per-process timers associated with a
 6777 clock other than CLOCK_REALTIME is implementation-defined.

6778 CS If the value of the CLOCK_REALTIME clock is set via *clock_settime()*, the new value of the clock
 6779 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)).

6805 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.

6808 ERRORS

6809 The *clock_getres()*, *clock_gettime()*, and *clock_settime()* functions shall fail if:

6810 [EINVAL] The *clock_id* argument does not specify a known clock.

6811 The *clock_settime()* function shall fail if:

6812 [EINVAL] The *tp* argument to *clock_settime()* is outside the range for the given clock ID.

6813 [EINVAL] The *tp* argument specified a nanosecond value less than zero or greater than or equal to 1 000 million.

6815 MON [EINVAL] The value of the *clock_id* argument is CLOCK_MONOTONIC.

6816 The *clock_settime()* function may fail if:

6817 [EPERM] The requesting process does not have the appropriate privilege to set the specified clock.

6819 EXAMPLES

6820 None.

6821 APPLICATION USAGE

6822 These functions are part of the Timers option and need not be available on all implementations.

6823 Note that the absolute value of the monotonic clock is meaningless (because its origin is
6824 arbitrary), and thus there is no need to set it. Furthermore, realtime applications can rely on the
6825 fact that the value of this clock is never set and, therefore, that time intervals measured with this
6826 clock will not be affected by calls to *clock_settime()*.

6827 RATIONALE

6828 None.

6829 FUTURE DIRECTIONS

6830 None.

6831 SEE ALSO

6832 *clock_getcpuclockid()*, *clock_nanosleep()*, *ctime()*, *mq_timedreceive()*, *mq_timedsend()*, *nanosleep()*,
6833 *pthread_mutex_timedlock()*, *sem_timedwait()*, *time()*, *timer_create()*, *timer_gettime()*, the Base
6834 Definitions volume of IEEE Std 1003.1-2001, <time.h>

6835 CHANGE HISTORY

6836 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

6837 Issue 6

6838 The [ENOSYS] error condition has been removed as stubs need not be provided if an
6839 implementation does not support the Timers option.

6840 The APPLICATION USAGE section is added.

6841 The following changes were made to align with the IEEE P1003.1a draft standard:

- 6842 • Clarification is added of the effect of resetting the clock resolution.

6843 CPU-time clocks and the *clock_getcpuclockid()* function are added for alignment with
6844 IEEE Std 1003.1d-1999.

6845 The following changes are added for alignment with IEEE Std 1003.1j-2000:

- 6846 • The DESCRIPTION is updated as follows:
 - 6847 — The value returned by *clock_gettime()* for CLOCK_MONOTONIC is specified.
 - 6848 — The *clock_settime()* function failing for CLOCK_MONOTONIC is specified.
 - 6849 — The effects of *clock_settime()* on the *clock_nanosleep()* function with respect to
6850 CLOCK_REALTIME are specified.
- 6851 • An [EINVAL] error is added to the ERRORS section, indicating that *clock_settime()* fails for
6852 CLOCK_MONOTONIC.
- 6853 • The APPLICATION USAGE section notes that the CLOCK_MONOTONIC clock need not
6854 and shall not be set by *clock_settime()* since the absolute value of the CLOCK_MONOTONIC
6855 clock is meaningless.
- 6856 • The *clock_nanosleep()*, *mq_timedreceive()*, *mq_timedsend()*, *pthread_mutex_timedlock()*,
6857 *sem_timedwait()*, *timer_create()*, and *timer_gettime()* functions are added to the SEE ALSO
6858 section.

6859 NAME

6860 clock_nanosleep — high resolution sleep with specifiable clock (**ADVANCED REALTIME**)

6861 SYNOPSIS

```
6862 CS #include <time.h>
6863     int clock_nanosleep(clockid_t clock_id, int flags,
6864         const struct timespec *rqtp, struct timespec *rmtp);
```

6866 DESCRIPTION

6867 If the flag `TIMER_ABSTIME` is not set in the `flags` argument, the `clock_nanosleep()` function shall
6868 cause the current thread to be suspended from execution until either the time interval specified
6869 by the `rqtp` argument has elapsed, or a signal is delivered to the calling thread and its action is to
6870 invoke a signal-catching function, or the process is terminated. The clock used to measure the
6871 time shall be the clock specified by `clock_id`.

6872 If the flag `TIMER_ABSTIME` is set in the `flags` argument, the `clock_nanosleep()` function shall
6873 cause the current thread to be suspended from execution until either the time value of the clock
6874 specified by `clock_id` reaches the absolute time specified by the `rqtp` argument, or a signal is
6875 delivered to the calling thread and its action is to invoke a signal-catching function, or the
6876 process is terminated. If, at the time of the call, the time value specified by `rqtp` is less than or
6877 equal to the time value of the specified clock, then `clock_nanosleep()` shall return immediately
6878 and the calling process shall not be suspended.

6879 The suspension time caused by this function may be longer than requested because the
6880 argument value is rounded up to an integer multiple of the sleep resolution, or because of the
6881 scheduling of other activity by the system. But, except for the case of being interrupted by a
6882 signal, the suspension time for the relative `clock_nanosleep()` function (that is, with the
6883 `TIMER_ABSTIME` flag not set) shall not be less than the time interval specified by `rqtp`, as
6884 measured by the corresponding clock. The suspension for the absolute `clock_nanosleep()` function
6885 (that is, with the `TIMER_ABSTIME` flag set) shall be in effect at least until the value of the
6886 corresponding clock reaches the absolute time specified by `rqtp`, except for the case of being
6887 interrupted by a signal.

6888 The use of the `clock_nanosleep()` function shall have no effect on the action or blockage of any
6889 signal.

6890 The `clock_nanosleep()` function shall fail if the `clock_id` argument refers to the CPU-time clock of
6891 the calling thread. It is unspecified whether `clock_id` values of other CPU-time clocks are
6892 allowed.

6893 RETURN VALUE

6894 If the `clock_nanosleep()` function returns because the requested time has elapsed, its return value
6895 shall be zero.

6896 If the `clock_nanosleep()` function returns because it has been interrupted by a signal, it shall return
6897 the corresponding error value. For the relative `clock_nanosleep()` function, if the `rmtp` argument is
6898 non-NULL, the `timespec` structure referenced by it shall be updated to contain the amount of
6899 time remaining in the interval (the requested time minus the time actually slept). If the `rmtp`
6900 argument is NULL, the remaining time is not returned. The absolute `clock_nanosleep()` function
6901 has no effect on the structure referenced by `rmtp`.

6902 If `clock_nanosleep()` fails, it shall return the corresponding error value.

6903 ERRORS

6904 The *clock_nanosleep()* function shall fail if:

6905 [EINVAL] The *clock_nanosleep()* function was interrupted by a signal.

6906 [EINVAL] The *rqtp* argument specified a nanosecond value less than zero or greater than
6907 or equal to 1 000 million; or the TIMER_ABSTIME flag was specified in flags
6908 and the *rqtp* argument is outside the range for the clock specified by *clock_id*;
6909 or the *clock_id* argument does not specify a known clock, or specifies the
6910 CPU-time clock of the calling thread.

6911 [ENOTSUP] The *clock_id* argument specifies a clock for which *clock_nanosleep()* is not
6912 supported, such as a CPU-time clock.

6913 EXAMPLES

6914 None.

6915 APPLICATION USAGE

6916 Calling *clock_nanosleep()* with the value TIMER_ABSTIME not set in the *flags* argument and with
6917 a *clock_id* of CLOCK_REALTIME is equivalent to calling *nanosleep()* with the same *rqtp* and *rmtlp*
6918 arguments.

6919 RATIONALE

6920 The *nanosleep()* function specifies that the system-wide clock CLOCK_REALTIME is used to
6921 measure the elapsed time for this time service. However, with the introduction of the monotonic
6922 clock CLOCK_MONOTONIC a new relative sleep function is needed to allow an application to
6923 take advantage of the special characteristics of this clock.

6924 There are many applications in which a process needs to be suspended and then activated
6925 multiple times in a periodic way; for example, to poll the status of a non-interrupting device or
6926 to refresh a display device. For these cases, it is known that precise periodic activation cannot be
6927 achieved with a relative *sleep()* or *nanosleep()* function call. Suppose, for example, a periodic
6928 process that is activated at time *T*₀, executes for a while, and then wants to suspend itself until
6929 time *T*₀+*T*, the period being *T*. If this process wants to use the *nanosleep()* function, it must first
6930 call *clock_gettime()* to get the current time, then calculate the difference between the current time
6931 and *T*₀+*T* and, finally, call *nanosleep()* using the computed interval. However, the process could
6932 be preempted by a different process between the two function calls, and in this case the interval
6933 computed would be wrong; the process would wake up later than desired. This problem would
6934 not occur with the absolute *clock_nanosleep()* function, since only one function call would be
6935 necessary to suspend the process until the desired time. In other cases, however, a relative sleep
6936 is needed, and that is why both functionalities are required.

6937 Although it is possible to implement periodic processes using the timers interface, this
6938 implementation would require the use of signals, and the reservation of some signal numbers. In
6939 this regard, the reasons for including an absolute version of the *clock_nanosleep()* function in
6940 IEEE Std 1003.1-2001 are the same as for the inclusion of the relative *nanosleep()*.

6941 It is also possible to implement precise periodic processes using *pthread_cond_timedwait()*, in
6942 which an absolute timeout is specified that takes effect if the condition variable involved is
6943 never signaled. However, the use of this interface is unnatural, and involves performing other
6944 operations on mutexes and condition variables that imply an unnecessary overhead.
6945 Furthermore, *pthread_cond_timedwait()* is not available in implementations that do not support
6946 threads.

6947 Although the interface of the relative and absolute versions of the new high resolution sleep
6948 service is the same *clock_nanosleep()* function, the *rmtlp* argument is only used in the relative
6949 sleep. This argument is needed in the relative *clock_nanosleep()* function to reissue the function

6950 call if it is interrupted by a signal, but it is not needed in the absolute *clock_nanosleep()* function
6951 call; if the call is interrupted by a signal, the absolute *clock_nanosleep()* function can be invoked
6952 again with the same *rqtp* argument used in the interrupted call.

6953 **FUTURE DIRECTIONS**

6954 None.

6955 **SEE ALSO**

6956 *clock_getres()*, *nanosleep()*, *pthread_cond_timedwait()*, *sleep()*, the Base Definitions volume of
6957 IEEE Std 1003.1-2001, <**time.h**>

6958 **CHANGE HISTORY**

6959 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

6960 NAME

6961 **clock_settime** — clock and timer functions (**REALTIME**)

6962 SYNOPSIS

6963 TMR `#include <time.h>`

6964 `int clock_settime(clockid_t clock_id, const struct timespec *tp);`

6965

6966 DESCRIPTION

6967 Refer to *clock_getres()*.

6968 NAME

6969 clog, clogf, clogl — complex natural logarithm functions

6970 SYNOPSIS

```
6971 #include <complex.h>
6972 double complex clog(double complex z);
6973 float complex clogf(float complex z);
6974 long double complex clogl(long double complex z);
```

6975 DESCRIPTION

6976 CX The functionality described on this reference page is aligned with the ISO C standard. Any
6977 conflict between the requirements described here and the ISO C standard is unintentional. This
6978 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

6979 These functions shall compute the complex natural (base e) logarithm of z , with a branch cut
6980 along the negative real axis.

6981 RETURN VALUE

6982 These functions shall return the complex natural logarithm value, in the range of a strip
6983 mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary
6984 axis.

6985 ERRORS

6986 No errors are defined.

6987 EXAMPLES

6988 None.

6989 APPLICATION USAGE

6990 None.

6991 RATIONALE

6992 None.

6993 FUTURE DIRECTIONS

6994 None.

6995 SEE ALSO

6996 *cexp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

6997 CHANGE HISTORY

6998 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

6999 NAME

7000 close — close a file descriptor

7001 SYNOPSIS

```
7002 #include <unistd.h>
7003 int close(int fildes);
```

7004 DESCRIPTION

7005 The *close()* function shall deallocate the file descriptor indicated by *fildes*. To deallocate means
 7006 to make the file descriptor available for return by subsequent calls to *open()* or other functions
 7007 that allocate file descriptors. All outstanding record locks owned by the process on the file
 7008 associated with the file descriptor shall be removed (that is, unlocked).

7009 If *close()* is interrupted by a signal that is to be caught, it shall return -1 with *errno* set to [EINTR]
 7010 and the state of *fildes* is unspecified. If an I/O error occurred while reading from or writing to the
 7011 file system during *close()*, it may return -1 with *errno* set to [EIO]; if this error is returned, the
 7012 state of *fildes* is unspecified.

7013 When all file descriptors associated with a pipe or FIFO special file are closed, any data
 7014 remaining in the pipe or FIFO shall be discarded.

7015 When all file descriptors associated with an open file description have been closed, the open file
 7016 description shall be freed.

7017 If the link count of the file is 0, when all file descriptors associated with the file are closed, the
 7018 space occupied by the file shall be freed and the file shall no longer be accessible.

7019 XSR If a STREAMS-based *fildes* is closed and the calling process was previously registered to receive
 7020 a SIGPOLL signal for events associated with that STREAM, the calling process shall be
 7021 unregistered for events associated with the STREAM. The last *close()* for a STREAM shall cause
 7022 the STREAM associated with *fildes* to be dismantled. If O_NONBLOCK is not set and there have
 7023 been no signals posted for the STREAM, and if there is data on the module's write queue, *close()*
 7024 shall wait for an unspecified time (for each module and driver) for any output to drain before
 7025 dismantling the STREAM. The time delay can be changed via an I_SETCLTIME *ioctl()* request. If
 7026 the O_NONBLOCK flag is set, or if there are any pending signals, *close()* shall not wait for
 7027 output to drain, and shall dismantle the STREAM immediately.

7028 If the implementation supports STREAMS-based pipes, and *fildes* is associated with one end of a
 7029 pipe, the last *close()* shall cause a hangup to occur on the other end of the pipe. In addition, if the
 7030 other end of the pipe has been named by *fattach()*, then the last *close()* shall force the named end
 7031 to be detached by *fdetach()*. If the named end has no open file descriptors associated with it and
 7032 gets detached, the STREAM associated with that end shall also be dismantled.

7033 XSI If *fildes* refers to the master side of a pseudo-terminal, and this is the last close, a SIGHUP signal
 7034 shall be sent to the controlling process, if any, for which the slave side of the pseudo-terminal is
 7035 the controlling terminal. It is unspecified whether closing the master side of the pseudo-terminal
 7036 flushes all queued input and output. 1 1

7037 XSR If *fildes* refers to the slave side of a STREAMS-based pseudo-terminal, a zero-length message
 7038 may be sent to the master.

7039 AIO When there is an outstanding cancelable asynchronous I/O operation against *fildes* when *close()*
 7040 is called, that I/O operation may be canceled. An I/O operation that is not canceled completes
 7041 as if the *close()* operation had not yet occurred. All operations that are not canceled shall
 7042 complete as if the *close()* blocked until the operations completed. The *close()* operation itself
 7043 need not block awaiting such I/O completion. Whether any I/O operation is canceled, and
 7044 which I/O operation may be canceled upon *close()*, is implementation-defined.

7045 MF|SHM If a shared memory object or a memory mapped file remains referenced at the last close (that is, a process has it mapped), then the entire contents of the memory object shall persist until the memory object becomes unreferenced. If this is the last close of a shared memory object or a memory mapped file and the close results in the memory object becoming unreferenced, and the memory object has been unlinked, then the memory object shall be removed.

7050 If *fdes* refers to a socket, *close()* shall cause the socket to be destroyed. If the socket is in connection-mode, and the SO_LINGER option is set for the socket with non-zero linger time, and the socket has untransmitted data, then *close()* shall block for up to the current linger interval until all data is transmitted.

7054 RETURN VALUE

7055 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to indicate the error.

7057 ERRORS

7058 The *close()* function shall fail if:

7059 [EBADF] The *fdes* argument is not a valid file descriptor.

7060 [EINTR] The *close()* function was interrupted by a signal.

7061 The *close()* function may fail if:

7062 [EIO] An I/O error occurred while reading from or writing to the file system.

7063 EXAMPLES

7064 Reassigning a File Descriptor

7065 The following example closes the file descriptor associated with standard output for the current process, re-assigns standard output to a new file descriptor, and closes the original file descriptor to clean up. This example assumes that the file descriptor 0 (which is the descriptor for standard input) is not closed.

```
7069 #include <unistd.h>
7070 ...
7071 int pfd;
7072 ...
7073 close(1);
7074 dup(pfd);
7075 close(pfd);
7076 ...
```

7077 Incidentally, this is exactly what could be achieved using:

```
7078 dup2(pfd, 1);
7079 close(pfd);
```

7080 Closing a File Descriptor

7081 In the following example, *close()* is used to close a file descriptor after an unsuccessful attempt is made to associate that file descriptor with a stream.

```
7083 #include <stdio.h>
7084 #include <unistd.h>
7085 #include <stdlib.h>
```

```
7086     #define LOCKFILE "/etc/ptmp"
7087     ...
7088     int pfd;
7089     FILE *fpfd;
7090     ...
7091     if ((fpfd = fdopen (pfd, "w")) == NULL) {
7092         close(pfd);
7093         unlink(LOCKFILE);
7094         exit(1);
7095     }
7096     ...
7097 
```

APPLICATION USAGE

7098 An application that had used the *stdio* routine *fopen()* to open a file should use the
7099 corresponding *fclose()* routine rather than *close()*. Once a file is closed, the file descriptor no
7100 longer exists, since the integer corresponding to it no longer refers to a file.

RATIONALE

7102 The use of interruptible device close routines should be discouraged to avoid problems with the
7103 implicit closes of file descriptors by *exec* and *exit()*. This volume of IEEE Std 1003.1-2001 only
7104 intends to permit such behavior by specifying the [EINTR] error condition.

FUTURE DIRECTIONS

7106 None.

SEE ALSO

7108 Section 2.6 (on page 38), *fattach()*, *fclose()*, *fdetach()*, *fopen()*, *ioctl()*, *open()*, the Base Definitions
7109 volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY

7111 First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

7113 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Issue 6

7115 The DESCRIPTION related to a STREAMS-based file or pseudo-terminal is marked as part of the
7116 XSI STREAMS Option Group.

7117 The following new requirements on POSIX implementations derive from alignment with the
7118 Single UNIX Specification:

- 7119 • The [EIO] error condition is added as an optional error.
- 7120 • The DESCRIPTION is updated to describe the state of the *fildes* file descriptor as unspecified
7121 if an I/O error occurs and an [EIO] error condition is returned.

7122 Text referring to sockets is added to the DESCRIPTION.

7123 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
7124 shared memory objects and memory mapped files (and not typed memory objects) are the types
7125 of memory objects to which the paragraph on last closes applies.

7126 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/12 is applied, correcting the XSH shaded 1
7127 text relating to the master side of a pseudo-terminal. The reason for the change is that the 1
7128 behavior of pseudo-terminals and regular terminals should be as much alike as possible in this 1
7129 case; the change achieves that and matches historical behavior. 1

7130 NAME

7131 closedir — close a directory stream

7132 SYNOPSIS

7133 #include <dirent.h>

7134 int closedir(DIR *dirp);

7135 DESCRIPTION

7136 The *closedir()* function shall close the directory stream referred to by the argument *dirp*. Upon
7137 return, the value of *dirp* may no longer point to an accessible object of the type **DIR**. If a file
7138 descriptor is used to implement type **DIR**, that file descriptor shall be closed.

7139 RETURN VALUE

7140 Upon successful completion, *closedir()* shall return 0; otherwise, -1 shall be returned and *errno*
7141 set to indicate the error.

7142 ERRORS

7143 The *closedir()* function may fail if:

7144 [EBADF] The *dirp* argument does not refer to an open directory stream.

7145 [EINTR] The *closedir()* function was interrupted by a signal.

7146 EXAMPLES**7147 Closing a Directory Stream**

7148 The following program fragment demonstrates how the *closedir()* function is used.

```
7149       ...
7150       DIR *dir;
7151       struct dirent *dp;
7152       ...
7153       if ((dir = opendir (".")) == NULL) {
7154       ...
7155       }
7156       while ((dp = readdir (dir)) != NULL) {
7157       ...
7158       }
7159       closedir(dir);
7160       ...
```

7161 APPLICATION USAGE

7162 None.

7163 RATIONALE

7164 None.

7165 FUTURE DIRECTIONS

7166 None.

7167 SEE ALSO

7168 *opendir()*, the Base Definitions volume of IEEE Std 1003.1-2001, <dirent.h>

7169 **CHANGE HISTORY**

7170 First released in Issue 2.

7171 **Issue 6**

7172 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

7173 The following new requirements on POSIX implementations derive from alignment with the
7174 Single UNIX Specification:

- 7175 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
7176 required for conforming implementations of previous POSIX specifications, it was not
7177 required for UNIX applications.
- 7178 • The [EINTR] error condition is added as an optional error condition.

7179 **NAME**

7180 closelog, openlog, setlogmask, syslog — control system log

7181 **SYNOPSIS**

```
7182 XSI #include <syslog.h>
7183
7184     void closelog(void);
7185     void openlog(const char *ident, int logopt, int facility);
7186     int setlogmask(int maskpri);
7187     void syslog(int priority, const char *message, ... /* arguments */);
```

7188 **DESCRIPTION**

7189 The *syslog()* function shall send a message to an implementation-defined logging facility, which
7190 may log it in an implementation-defined system log, write it to the system console, forward it to
7191 a list of users, or forward it to the logging facility on another host over the network. The logged
7192 message shall include a message header and a message body. The message header contains at
7193 least a timestamp and a tag string.

7194 The message body is generated from the *message* and following arguments in the same manner
7195 as if these were arguments to *printf()*, except that the additional conversion specification %m
7196 shall be recognized; it shall convert no arguments, shall cause the output of the error message
7197 string associated with the value of *errno* on entry to *syslog()*, and may be mixed with argument
7198 specifications of the "%n\$" form. If a complete conversion specification with the m conversion
7199 specifier character is not just %m, the behavior is undefined. A trailing <newline> may be added
7200 if needed.

7201 Values of the *priority* argument are formed by OR'ing together a severity-level value and an
7202 optional facility value. If no facility value is specified, the current default facility value is used.

7203 Possible values of severity level include:

7204 LOG_EMERG	A panic condition.
7205 LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system 7206 database.
7207 LOG_CRIT	Critical conditions, such as hard device errors.
7208 LOG_ERR	Errors.
7209 LOG_WARNING	Warning messages.
7211 LOG_NOTICE	Conditions that are not error conditions, but that may require special 7212 handling.
7213 LOG_INFO	Informational messages.
7214 LOG_DEBUG	Messages that contain information normally of use only when debugging a 7215 program.

7216 The facility indicates the application or system component generating the message. Possible
7217 facility values include:

7218 LOG_USER	Messages generated by arbitrary processes. This is the default facility 7219 identifier if none is specified.
7220 LOG_LOCAL0	Reserved for local use.

7221 LOG_LOCAL1 Reserved for local use.
7222 LOG_LOCAL2 Reserved for local use.
7223 LOG_LOCAL3 Reserved for local use.
7224 LOG_LOCAL4 Reserved for local use.
7225 LOG_LOCAL5 Reserved for local use.
7226 LOG_LOCAL6 Reserved for local use.
7227 LOG_LOCAL7 Reserved for local use.

7228 The *openlog()* function shall set process attributes that affect subsequent calls to *syslog()*. The *ident* argument is a string that is prepended to every message. The *logopt* argument indicates logging options. Values for *logopt* are constructed by a bitwise-inclusive OR of zero or more of the following:

7232 LOG_PID Log the process ID with each message. This is useful for identifying specific
7233 processes.
7234 LOG_CONS Write messages to the system console if they cannot be sent to the logging
7235 facility. The *syslog()* function ensures that the process does not acquire the
7236 console as a controlling terminal in the process of writing the message.
7237 LOG_NDELAY Open the connection to the logging facility immediately. Normally the open is
7238 delayed until the first message is logged. This is useful for programs that need
7239 to manage the order in which file descriptors are allocated.
7240 LOG_ODELAY Delay open until *syslog()* is called.
7241 LOG_NOWAIT Do not wait for child processes that may have been created during the course
7242 of logging the message. This option should be used by processes that enable
7243 notification of child termination using SIGCHLD, since *syslog()* may
7244 otherwise block waiting for a child whose exit status has already been
7245 collected.

7246 The *facility* argument encodes a default facility to be assigned to all messages that do not have
7247 an explicit facility already encoded. The initial default facility is LOG_USER.

7248 The *openlog()* and *syslog()* functions may allocate a file descriptor. It is not necessary to call
7249 *openlog()* prior to calling *syslog()*.

7250 The *closelog()* function shall close any open file descriptors allocated by previous calls to
7251 *openlog()* or *syslog()*.

7252 The *setlogmask()* function shall set the log priority mask for the current process to *maskpri* and
7253 return the previous mask. If the *maskpri* argument is 0, the current log mask is not modified.
7254 Calls by the current process to *syslog()* with a priority not set in *maskpri* shall be rejected. The
7255 default log mask allows all priorities to be logged. A call to *openlog()* is not required prior to
7256 calling *setlogmask()*.

7257 Symbolic constants for use as values of the *logopt*, *facility*, *priority*, and *maskpri* arguments are
7258 defined in the <syslog.h> header.

7259 RETURN VALUE

7260 The *setlogmask()* function shall return the previous log priority mask. The *closelog()*, *openlog()*,
7261 and *syslog()* functions shall not return a value.

7262 ERRORS

7263 No errors are defined.

7264 EXAMPLES**7265 Using openlog()**

7266 The following example causes subsequent calls to *syslog()* to log the process ID with each
7267 message, and to write messages to the system console if they cannot be sent to the logging
7268 facility.

```
7269 #include <syslog.h>
7270
7271     char *ident = "Process demo";
7272     int logopt = LOG_PID | LOG_CONS;
7273     int facility = LOG_USER;
7274
7275     ...
7276     openlog(ident, logopt, facility);
```

7275 Using setlogmask()

7276 The following example causes subsequent calls to *syslog()* to accept error messages, and to reject 1
7277 all other messages. 1

```
7278 #include <syslog.h>
7279
7280     int result;
7281     int mask = LOG_MASK (LOG_ERR);
7282
7283     ...
7284     result = setlogmask(mask);
```

7283 Using syslog

7284 The following example sends the message "This is a message" to the default logging
7285 facility, marking the message as an error message generated by random processes.

```
7286 #include <syslog.h>
7287
7288     char *message = "This is a message";
7289     int priority = LOG_ERR | LOG_USER;
7290
7291     ...
7292     syslog(priority, message);
```

7291 APPLICATION USAGE

7292 None.

7293 RATIONALE

7294 None.

7295 FUTURE DIRECTIONS

7296 None.

7297 SEE ALSO

7298 *printf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <syslog.h>

7299 CHANGE HISTORY

7300 First released in Issue 4, Version 2.

7301 Issue 5

7302 Moved from X/OPEN UNIX extension to BASE.

7303 Issue 6

7304 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/13 is applied, correcting the EXAMPLES 1
7305 section. 1

7306 NAME

7307 confstr — get configurable variables

7308 SYNOPSIS

```
7309     #include <unistd.h>
7310
7311     size_t confstr(int name, char *buf, size_t len);
```

7311 DESCRIPTION

7312 The *confstr()* function shall return configuration-defined string values. Its use and purpose are
 7313 similar to *sysconf()*, but it is used where string values rather than numeric values are returned.

7314 The *name* argument represents the system variable to be queried. The implementation shall
 7315 support the following name values, defined in <**unistd.h**>. It may support others:

```
7316     _CS_PATH
7317     _CS_POSIX_V6_ILP32_OFF32_CFLAGS
7318     _CS_POSIX_V6_ILP32_OFF32_LDFLAGS
7319     _CS_POSIX_V6_ILP32_OFF32_LIBS
7320     _CS_POSIX_V6_ILP32_OFFBIG_CFLAGS
7321     _CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS
7322     _CS_POSIX_V6_ILP32_OFFBIG_LIBS
7323     _CS_POSIX_V6_LP64_OFF64_CFLAGS
7324     _CS_POSIX_V6_LP64_OFF64_LDFLAGS
7325     _CS_POSIX_V6_LP64_OFF64_LIBS
7326     _CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS
7327     _CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS
7328     _CS_POSIX_V6_LPBIG_OFFBIG_LIBS
7329     _CS_POSIX_V6_WIDTH_RESTRICTED_ENVS
7330 XSI     _CS_XBS5_ILP32_OFF32_CFLAGS (LEGACY)
7331     _CS_XBS5_ILP32_OFF32_LDFLAGS (LEGACY)
7332     _CS_XBS5_ILP32_OFF32_LIBS (LEGACY)
7333     _CS_XBS5_ILP32_OFF32_LINTFLAGS (LEGACY)
7334     _CS_XBS5_ILP32_OFFBIG_CFLAGS (LEGACY)
7335     _CS_XBS5_ILP32_OFFBIG_LDFLAGS (LEGACY)
7336     _CS_XBS5_ILP32_OFFBIG_LIBS (LEGACY)
7337     _CS_XBS5_ILP32_OFFBIG_LINTFLAGS (LEGACY)
7338     _CS_XBS5_LP64_OFF64_CFLAGS (LEGACY)
7339     _CS_XBS5_LP64_OFF64_LDFLAGS (LEGACY)
7340     _CS_XBS5_LP64_OFF64_LIBS (LEGACY)
7341     _CS_XBS5_LP64_OFF64_LINTFLAGS (LEGACY)
7342     _CS_XBS5_LPBIG_OFFBIG_CFLAGS (LEGACY)
7343     _CS_XBS5_LPBIG_OFFBIG_LDFLAGS (LEGACY)
7344     _CS_XBS5_LPBIG_OFFBIG_LIBS (LEGACY)
7345     _CS_XBS5_LPBIG_OFFBIG_LINTFLAGS (LEGACY)
```

7347 If *len* is not 0, and if *name* has a configuration-defined value, *confstr()* shall copy that value into
 7348 the *len*-byte buffer pointed to by *buf*. If the string to be returned is longer than *len* bytes,
 7349 including the terminating null, then *confstr()* shall truncate the string to *len*-1 bytes and null-
 7350 terminate the result. The application can detect that the string was truncated by comparing the
 7351 value returned by *confstr()* with *len*.

7352 If *len* is 0 and *buf* is a null pointer, then *confstr()* shall still return the integer value as defined
 7353 below, but shall not return a string. If *len* is 0 but *buf* is not a null pointer, the result is
 7354 unspecified.

7355 If the implementation supports the POSIX shell option, the string stored in *buf* after a call to:
7356 `confstr(_CS_PATH, buf, sizeof(buf))`
7357 can be used as a value of the *PATH* environment variable that accesses all of the standard
7358 utilities of IEEE Std 1003.1-2001, if the return value is less than or equal to *sizeof(buf)*.

7359 RETURN VALUE

7360 If *name* has a configuration-defined value, *confstr()* shall return the size of buffer that would be
7361 needed to hold the entire configuration-defined value including the terminating null. If this
7362 return value is greater than *len*, the string returned in *buf* is truncated.
7363 If *name* is invalid, *confstr()* shall return 0 and set *errno* to indicate the error.
7364 If *name* does not have a configuration-defined value, *confstr()* shall return 0 and leave *errno*
7365 unchanged.

7366 ERRORS

7367 The *confstr()* function shall fail if:

7368 [EINVAL] The value of the *name* argument is invalid.

7369 EXAMPLES

7370 None.

7371 APPLICATION USAGE

7372 An application can distinguish between an invalid *name* parameter value and one that
7373 corresponds to a configurable variable that has no configuration-defined value by checking if
7374 *errno* is modified. This mirrors the behavior of *sysconf()*.

7375 The original need for this function was to provide a way of finding the configuration-defined
7376 default value for the environment variable *PATH*. Since *PATH* can be modified by the user to
7377 include directories that could contain utilities replacing the standard utilities in the Shell and
7378 Utilities volume of IEEE Std 1003.1-2001, applications need a way to determine the system-
7379 supplied *PATH* environment variable value that contains the correct search path for the standard
7380 utilities.

7381 An application could use:

7382 `confstr(name, (char *)NULL, (size_t)0)`

7383 to find out how big a buffer is needed for the string value; use *malloc()* to allocate a buffer to
7384 hold the string; and call *confstr()* again to get the string. Alternately, it could allocate a fixed,
7385 static buffer that is big enough to hold most answers (perhaps 512 or 1 024 bytes), but then use
7386 *malloc()* to allocate a larger buffer if it finds that this is too small.

7387 RATIONALE

7388 Application developers can normally determine any configuration variable by means of reading
7389 from the stream opened by a call to:

7390 `popen("command -p getconf variable", "r");`

7391 The *confstr()* function with a *name* argument of *_CS_PATH* returns a string that can be used as a
7392 *PATH* environment variable setting that will reference the standard shell and utilities as
7393 described in the Shell and Utilities volume of IEEE Std 1003.1-2001.

7394 The *confstr()* function copies the returned string into a buffer supplied by the application instead
7395 of returning a pointer to a string. This allows a cleaner function in some implementations (such
7396 as those with lightweight threads) and resolves questions about when the application must copy
7397 the string returned.

7398 **FUTURE DIRECTIONS**

7399 None.

7400 **SEE ALSO**7401 *pathconf()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>, the Shell
7402 and Utilities volume of IEEE Std 1003.1-2001, *c99*7403 **CHANGE HISTORY**

7404 First released in Issue 4. Derived from the ISO POSIX-2 standard.

7405 **Issue 5**7406 A table indicating the permissible values of *name* is added to the DESCRIPTION. All those
7407 marked EX are new in this issue.7408 **Issue 6**7409 The Open Group Corrigendum U033/7 is applied. The return value for the case returning the
7410 size of the buffer now explicitly states that this includes the terminating null.7411 The following new requirements on POSIX implementations derive from alignment with the
7412 Single UNIX Specification:

- 7413 • The DESCRIPTION is updated with new arguments which can be used to determine
7414 configuration strings for C compiler flags, linker/loader flags, and libraries for each different
7415 supported programming environment. This is a change to support data size neutrality.

7416 The following changes were made to align with the IEEE P1003.1a draft standard:

- 7417 • The DESCRIPTION is updated to include text describing how _CS_PATH can be used to
7418 obtain a PATH to access the standard utilities.

7419 The macros associated with the *c89* programming models are marked LEGACY and new
7420 equivalent macros associated with *c99* are introduced.

7421 NAME

7422 **conj, conjf, conjl** — complex conjugate functions

7423 SYNOPSIS

```
7424     #include <complex.h>
7425
7426     double complex conj(double complex z);
7427     float complex conjf(float complex z);
7428     long double complex conjl(long double complex z);
```

7428 DESCRIPTION

7429 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
7430 conflict between the requirements described here and the ISO C standard is unintentional. This
7431 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7432 These functions shall compute the complex conjugate of *z*, by reversing the sign of its imaginary
7433 part.

7434 RETURN VALUE

7435 These functions return the complex conjugate value.

7436 ERRORS

7437 No errors are defined.

7438 EXAMPLES

7439 None.

7440 APPLICATION USAGE

7441 None.

7442 RATIONALE

7443 None.

7444 FUTURE DIRECTIONS

7445 None.

7446 SEE ALSO

7447 **carg(), cimag(), cproj(), creal()**, the Base Definitions volume of IEEE Std 1003.1-2001,
7448 **<complex.h>**

7449 CHANGE HISTORY

7450 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7451 **NAME**

7452 connect — connect a socket

7453 **SYNOPSIS**

```
7454 #include <sys/socket.h>
7455 int connect(int socket, const struct sockaddr *address,
7456             socklen_t address_len);
```

7457 **DESCRIPTION**

7458 The *connect()* function shall attempt to make a connection on a socket. The function takes the
7459 following arguments:

7460	<i>socket</i>	Specifies the file descriptor associated with the socket.
7461	<i>address</i>	Points to a sockaddr structure containing the peer address. The length and 7462 format of the address depend on the address family of the socket.
7463	<i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> 7464 argument.

7465 If the socket has not already been bound to a local address, *connect()* shall bind it to an address
7466 which, unless the socket's address family is AF_UNIX, is an unused local address.

7467 If the initiating socket is not connection-mode, then *connect()* shall set the socket's peer address,
7468 and no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all
7469 datagrams are sent on subsequent *send()* functions, and limits the remote sender for subsequent
7470 *recv()* functions. If *address* is a null address for the protocol, the socket's peer address shall be
7471 reset.

7472 If the initiating socket is connection-mode, then *connect()* shall attempt to establish a connection
7473 to the address specified by the *address* argument. If the connection cannot be established
7474 immediately and O_NONBLOCK is not set for the file descriptor for the socket, *connect()* shall
7475 block for up to an unspecified timeout interval until the connection is established. If the timeout
7476 interval expires before the connection is established, *connect()* shall fail and the connection
7477 attempt shall be aborted. If *connect()* is interrupted by a signal that is caught while blocked
7478 waiting to establish a connection, *connect()* shall fail and set *errno* to [EINTR], but the connection
7479 request shall not be aborted, and the connection shall be established asynchronously.

7480 If the connection cannot be established immediately and O_NONBLOCK is set for the file
7481 descriptor for the socket, *connect()* shall fail and set *errno* to [EINPROGRESS], but the connection
7482 request shall not be aborted, and the connection shall be established asynchronously.
7483 Subsequent calls to *connect()* for the same socket, before the connection is established, shall fail
7484 and set *errno* to [EALREADY].

7485 When the connection has been established asynchronously, *select()* and *poll()* shall indicate that
7486 the file descriptor for the socket is ready for writing.

7487 The socket in use may require the process to have appropriate privileges to use the *connect()*
7488 function.

7489 **RETURN VALUE**

7490 Upon successful completion, *connect()* shall return 0; otherwise, -1 shall be returned and *errno*
7491 set to indicate the error.

7492 **ERRORS**

7493 The *connect()* function shall fail if:

7494 [EADDRNOTAVAIL]

7495 The specified address is not available from the local machine.

7496	[EAFNOSUPPORT]	The specified address is not a valid address for the address family of the specified socket.
7497		
7498		
7499	[EALREADY]	A connection request is already in progress for the specified socket.
7500	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
7501	[ECONNREFUSED]	
7502		The target address was not listening for connections or refused the connection request.
7503		
7504	[EINPROGRESS]	O_NONBLOCK is set for the file descriptor for the socket and the connection cannot be immediately established; the connection shall be established asynchronously.
7505		
7506		
7507	[EINTR]	The attempt to establish a connection was interrupted by delivery of a signal that was caught; the connection shall be established asynchronously.
7508		
7509	[EISCONN]	The specified socket is connection-mode and is already connected.
7510	[ENETUNREACH]	
7511		No route to the network is present.
7512	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
7513	[EPROTOTYPE]	The specified address has a different type than the socket bound to the specified peer address.
7514		
7515	[ETIMEDOUT]	The attempt to connect timed out before a connection was made.
7516		If the address family of the socket is AF_UNIX, then <i>connect()</i> shall fail if:
7517	[EIO]	An I/O error occurred while reading from or writing to the file system.
7518	[ELOOP]	A loop exists in symbolic links encountered during resolution of the pathname in <i>address</i> .
7519		
7520	[ENAMETOOLONG]	
7521		A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
7522		
7523	[ENOENT]	A component of the pathname does not name an existing file or the pathname is an empty string.
7524		
7525	[ENOTDIR]	A component of the path prefix of the pathname in <i>address</i> is not a directory.
7526		The <i>connect()</i> function may fail if:
7527	[EACCES]	Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
7528		
7529	[EADDRINUSE]	Attempt to establish a connection that uses addresses that are already in use.
7530	[ECONNRESET]	Remote host reset the connection request.
7531	[EHOSTUNREACH]	
7532		The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
7533		
7534	[EINVAL]	The <i>address_len</i> argument is not a valid length for the address family; or invalid address family in the <i>sockaddr</i> structure.
7535		

- 7536 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
7537 resolution of the pathname in *address*.
- 7538 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
7539 whose length exceeds {PATH_MAX}.
- 7541 [ENETDOWN] The local network interface used to reach the destination is down.
- 7542 [ENOBUFS] No buffer space is available.
- 7543 [EOPNOTSUPP] The socket is listening and cannot be connected.

7544 EXAMPLES

7545 None.

7546 APPLICATION USAGE

7547 If *connect()* fails, the state of the socket is unspecified. Conforming applications should close the
7548 file descriptor and create a new socket before attempting to reconnect.

7549 RATIONALE

7550 None.

7551 FUTURE DIRECTIONS

7552 None.

7553 SEE ALSO

7554 *accept()*, *bind()*, *close()*, *getsockname()*, *poll()*, *select()*, *send()*, *shutdown()*, *socket()*, the Base
7555 Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

7556 CHANGE HISTORY

7557 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

7558 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
7559 [ELOOP] error condition is added.

7560 NAME

7561 copysign, copysignf, copysignl — number manipulation function

7562 SYNOPSIS

```
7563        #include <math.h>
7564
7565        double copysign(double x, double y);
7566        float copysignf(float x, float y);
7567        long double copysignl(long double x, long double y);
```

7568 DESCRIPTION

7569 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7570 conflict between the requirements described here and the ISO C standard is unintentional. This
7571 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7572 These functions shall produce a value with the magnitude of *x* and the sign of *y*. On
7573 implementations that represent a signed zero but do not treat negative zero consistently in
7574 arithmetic operations, these functions regard the sign of zero as positive.

7575 RETURN VALUE

7576 Upon successful completion, these functions shall return a value with the magnitude of *x* and
7577 the sign of *y*.

7578 ERRORS

7579 No errors are defined.

7580 EXAMPLES

7581 None.

7582 APPLICATION USAGE

7583 None.

7584 RATIONALE

7585 None.

7586 FUTURE DIRECTIONS

7587 None.

7588 SEE ALSO

7589 *signbit()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**math.h**>

7590 CHANGE HISTORY

7591 First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

7591 NAME

7592 cos, cosf, cosl — cosine function

7593 SYNOPSIS

```
7594     #include <math.h>
7595
7596     double cos(double x);
7597     float cosf(float x);
7598     long double cosl(long double x);
```

7598 DESCRIPTION

7599 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7600 conflict between the requirements described here and the ISO C standard is unintentional. This
7601 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7602 These functions shall compute the cosine of their argument *x*, measured in radians.

7603 An application wishing to check for error situations should set *errno* to zero and call
7604 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
7605 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
7606 zero, an error has occurred.

7607 RETURN VALUE

7608 Upon successful completion, these functions shall return the cosine of *x*.

7609 MX If *x* is NaN, a NaN shall be returned.

7610 If *x* is ±0, the value 1.0 shall be returned.

7611 If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-
7612 defined value shall be returned.

7613 ERRORS

7614 These functions shall fail if:

7615 MX Domain Error The *x* argument is ±Inf.

7616 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
7617 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
7618 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
7619 shall be raised.

7620 EXAMPLES**7621 Taking the Cosine of a 45-Degree Angle**

```
7622     #include <math.h>
7623
7624     ...
7625     double radians = 45 * M_PI / 180;
7626     double result;
7627
7628     result = cos(radians);
```

7628 APPLICATION USAGE

7629 These functions may lose accuracy when their argument is near an odd multiple of $\pi/2$ or is far
7630 from 0.

7631 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
7632 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

7633 RATIONALE

7634 None.

7635 FUTURE DIRECTIONS

7636 None.

7637 SEE ALSO

7638 *acos()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *sin()*, *tan()*, the Base Definitions volume of
7639 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
7640 <**math.h**>

7641 CHANGE HISTORY

7642 First released in Issue 1. Derived from Issue 1 of the SVID.

7643 Issue 5

7644 The DESCRIPTION is updated to indicate how an application should check for an error. This
7645 text was previously published in the APPLICATION USAGE section.

7646 Issue 6

7647 The *cosf()* and *cosl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

7648 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
7649 revised to align with the ISO/IEC 9899:1999 standard.

7650 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
7651 marked.

7652 NAME

7653 cosh, coshf, coshl — hyperbolic cosine functions

7654 SYNOPSIS

```
7655     #include <math.h>
7656
7657     double cosh(double x);
7658     float coshf(float x);
7659     long double coshl(long double x);
```

7659 DESCRIPTION

7660 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7661 conflict between the requirements described here and the ISO C standard is unintentional. This
7662 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7663 These functions shall compute the hyperbolic cosine of their argument x .

7664 An application wishing to check for error situations should set $errno$ to zero and call
7665 $\text{feclearexcept}(\text{FE_ALL_EXCEPT})$ before calling these functions. On return, if $errno$ is non-zero or
7666 $\text{fetestexcept}(\text{FE_INVALID} \mid \text{FE_DIVBYZERO} \mid \text{FE_OVERFLOW} \mid \text{FE_UNDERFLOW})$ is non-
7667 zero, an error has occurred.

7668 RETURN VALUE

7669 Upon successful completion, these functions shall return the hyperbolic cosine of x .

7670 If the correct value would cause overflow, a range error shall occur and $\text{cosh}()$, $\text{coshf}()$, and
7671 $\text{coshl}()$ shall return the value of the macro `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`,
7672 respectively.

7673 MX If x is NaN, a NaN shall be returned.

7674 If x is ± 0 , the value 1.0 shall be returned.

7675 If x is $\pm\text{Inf}$, +Inf shall be returned.

7676 ERRORS

7677 These functions shall fail if:

7678 Range Error The result would cause an overflow.

7679 If the integer expression (`math_errhandling & MATH_ERRNO`) is non-zero,
7680 then $errno$ shall be set to [ERANGE]. If the integer expression
7681 (`math_errhandling & MATH_ERREXCEPT`) is non-zero, then the overflow
7682 floating-point exception shall be raised.

7683 EXAMPLES

7684 None.

7685 APPLICATION USAGE

7686 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling &`
7687 `MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

7688 For IEEE Std 754-1985 **double**, $710.5 < |x|$ implies that $\text{cosh}(x)$ has overflowed.

7689 RATIONALE

7690 None.

7691 FUTURE DIRECTIONS

7692 None.

7693 SEE ALSO

7694 *acosh()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *sinh()*, *tanh()*, the Base Definitions volume of
7695 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
7696 <math.h>

7697 CHANGE HISTORY

7698 First released in Issue 1. Derived from Issue 1 of the SVID.

7699 Issue 5

7700 The DESCRIPTION is updated to indicate how an application should check for an error. This
7701 text was previously published in the APPLICATION USAGE section.

7702 Issue 6

7703 The *coshf()* and *coshl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.
7704 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
7705 revised to align with the ISO/IEC 9899:1999 standard.
7706 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
7707 marked.

```
7708 NAME
7709         cosl — cosine function
7710 SYNOPSIS
7711         #include <math.h>
7712         long double cosl(long double x);
7713 DESCRIPTION
7714         Refer to cos().
```

7715 NAME

7716 `cpow, cpowf, cpowl — complex power functions`

7717 SYNOPSIS

```
7718        #include <complex.h>
7719        double complex cpow(double complex x, double complex y);
7720        float complex cpowf(float complex x, float complex y);
7721        long double complex cpowl(long double complex x,
7722                              long double complex y);
```

7723 DESCRIPTION

7724 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7725 conflict between the requirements described here and the ISO C standard is unintentional. This
7726 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7727 These functions shall compute the complex power function x^y , with a branch cut for the first
7728 parameter along the negative real axis.

7729 RETURN VALUE

7730 These functions shall return the complex power function value.

7731 ERRORS

7732 No errors are defined.

7733 EXAMPLES

7734 None.

7735 APPLICATION USAGE

7736 None.

7737 RATIONALE

7738 None.

7739 FUTURE DIRECTIONS

7740 None.

7741 SEE ALSO

7742 `cabs(), csqrt()`, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

7743 CHANGE HISTORY

7744 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7745 NAME

7746 *cproj*, *cprojf*, *cprojl* — complex projection functions

7747 SYNOPSIS

```
7748     #include <complex.h>
7749
7750     double complex cproj(double complex z);
7751     float complex cprojf(float complex z);
7752     long double complex cprojl(long double complex z);
```

7752 DESCRIPTION

7753 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7754 conflict between the requirements described here and the ISO C standard is unintentional. This
7755 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7756 These functions shall compute a projection of *z* onto the Riemann sphere: *z* projects to *z*, except
7757 that all complex infinities (even those with one infinite part and one NaN part) project to
7758 positive infinity on the real axis. If *z* has an infinite part, then *cproj(z)* shall be equivalent to:

```
7759     INFINITY + I * copysign(0.0, cimag(z))
```

7760 RETURN VALUE

7761 These functions shall return the value of the projection onto the Riemann sphere.

7762 ERRORS

7763 No errors are defined.

7764 EXAMPLES

7765 None.

7766 APPLICATION USAGE

7767 None.

7768 RATIONALE

7769 Two topologies are commonly used in complex mathematics: the complex plane with its
7770 continuum of infinities, and the Riemann sphere with its single infinity. The complex plane is
7771 better suited for transcendental functions, the Riemann sphere for algebraic functions. The
7772 complex types with their multiplicity of infinities provide a useful (though imperfect) model for
7773 the complex plane. The *cproj()* function helps model the Riemann sphere by mapping all
7774 infinities to one, and should be used just before any operation, especially comparisons, that
7775 might give spurious results for any of the other infinities. Note that a complex value with one
7776 infinite part and one NaN part is regarded as an infinity, not a NaN, because if one part is
7777 infinite, the complex value is infinite independent of the value of the other part. For the same
7778 reason, *cabs()* returns an infinity if its argument has an infinite part and a NaN part.

7779 FUTURE DIRECTIONS

7780 None.

7781 SEE ALSO

7782 *carg()*, *cimag()*, *conj()*, *creal()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

7783 CHANGE HISTORY

7784 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7785 NAME

7786 creal, crealf, creall — complex real functions

7787 SYNOPSIS

```
7788        #include <complex.h>
7789        double creal(double complex z);
7790        float crealf(float complex z);
7791        long double creall(long double complex z);
```

7792 DESCRIPTION

7793 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7794 conflict between the requirements described here and the ISO C standard is unintentional. This
7795 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7796 These functions shall compute the real part of *z*.

7797 RETURN VALUE

7798 These functions shall return the real part value.

7799 ERRORS

7800 No errors are defined.

7801 EXAMPLES

7802 None.

7803 APPLICATION USAGE

7804 For a variable *z* of type **complex**:

```
7805        z == creal(z) + cimag(z)*I
```

7806 RATIONALE

7807 None.

7808 FUTURE DIRECTIONS

7809 None.

7810 SEE ALSO

7811 *carg()*, *cimag()*, *conj()*, *cproj()*, the Base Definitions volume of IEEE Std 1003.1-2001,
7812 <**complex.h**>

7813 CHANGE HISTORY

7814 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7815 NAME

7816 creat — create a new file or rewrite an existing one

7817 SYNOPSIS

```
7818 OH #include <sys/stat.h>
7819 #include <fcntl.h>
7820 int creat(const char *path, mode_t mode);
```

7821 DESCRIPTION

7822 The function call:

```
7823     creat(path, mode)
7824 shall be equivalent to:
7825     open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

7826 RETURN VALUE

7827 Refer to *open()*.

7828 ERRORS

7829 Refer to *open()*.

7830 EXAMPLES**7831 Creating a File**

7832 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.

```
7835 #include <fcntl.h>
7836 ...
7837 int fd;
7838 mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
7839 char *filename = "/tmp/file";
7840 ...
7841 fd = creat(filename, mode);
7842 ...
```

7843 APPLICATION USAGE

7844 None.

7845 RATIONALE

7846 The *creat()* function is redundant. Its services are also provided by the *open()* function. It has
7847 been included primarily for historical purposes since many existing applications depend on it. It
7848 is best considered a part of the C binding rather than a function that should be provided in other
7849 languages.

7850 FUTURE DIRECTIONS

7851 None.

7852 SEE ALSO

7853 *open()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<sys/stat.h>**,
7854 **<sys/types.h>**

7855 **CHANGE HISTORY**

7856 First released in Issue 1. Derived from Issue 1 of the SVID.

7857 **Issue 6**

7858 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

7859 The following new requirements on POSIX implementations derive from alignment with the
7860 Single UNIX Specification:

- 7861 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
-
- 7862 required for conforming implementations of previous POSIX specifications, it was not
-
- 7863 required for UNIX applications.

7864 **NAME**7865 crypt — string encoding function (**CRYPT**)7866 **SYNOPSIS**

7867 XSI #include <unistd.h>

7868 char *crypt(const char *key, const char *salt);

7869

7870 **DESCRIPTION**7871 The *crypt()* function is a string encoding function. The algorithm is implementation-defined.7872 The *key* argument points to a string to be encoded. The *salt* argument is a string chosen from the set:7874 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
7875 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
7876 0 1 2 3 4 5 6 7 8 9 . /

7877 The first two characters of this string may be used to perturb the encoding algorithm.

7878 The return value of *crypt()* points to static data that is overwritten by each call.7879 The *crypt()* function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.7881 **RETURN VALUE**7882 Upon successful completion, *crypt()* shall return a pointer to the encoded string. The first two characters of the returned value shall be those of the *salt* argument. Otherwise, it shall return a null pointer and set *errno* to indicate the error.7885 **ERRORS**7886 The *crypt()* function shall fail if:

7887 [ENOSYS] The functionality is not supported on this implementation.

7888 **EXAMPLES**7889 **Encoding Passwords**7890 The following example finds a user database entry matching a particular user name and changes the current password to a new password. The *crypt()* function generates an encoded version of each password. The first call to *crypt()* produces an encoded version of the old password; that encoded password is then compared to the password stored in the user database. The second call to *crypt()* encodes the new password before it is stored.7895 The *putpwent()* function, used in the following example, is not part of IEEE Std 1003.1-2001.7896 #include <unistd.h>
7897 #include <pwd.h>
7898 #include <string.h>
7899 #include <stdio.h>
7900 ...
7901 int valid_change;
7902 int pfd; /* Integer for file descriptor returned by open(). */
7903 FILE *fpfd; /* File pointer for use in putpwent(). */
7904 struct passwd *p;
7905 char user[100];
7906 char oldpasswd[100];
7907 char newpasswd[100];

```
7908     char savepasswd[100];
7909     ...
7910     valid_change = 0;
7911     while ((p = getpwent()) != NULL) {
7912         /* Change entry if found. */
7913         if (strcmp(p->pw_name, user) == 0) {
7914             if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
7915                 strcpy(savepasswd, crypt(newpasswd, user));
7916                 p->pw_passwd = savepasswd;
7917                 valid_change = 1;
7918             }
7919             else {
7920                 fprintf(stderr, "Old password is not valid\n");
7921             }
7922         }
7923         /* Put passwd entry into ptmp. */
7924         putpwent(p, fpfd);
7925     }
```

7926 APPLICATION USAGE

7927 The values returned by this function need not be portable among XSI-conformant systems.

7928 RATIONALE

7929 None.

7930 FUTURE DIRECTIONS

7931 None.

7932 SEE ALSO

7933 *encrypt()*, *setkey()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

7934 CHANGE HISTORY

7935 First released in Issue 1. Derived from Issue 1 of the SVID.

7936 Issue 5

7937 Normative text previously in the APPLICATION USAGE section is moved to the
7938 DESCRIPTION.

7939 NAME

7940 csin, csinf, csinl — complex sine functions

7941 SYNOPSIS

```
7942        #include <complex.h>
7943        double complex csin(double complex z);
7944        float complex csinf(float complex z);
7945        long double complex csinl(long double complex z);
```

7946 DESCRIPTION

7947 CX The functionality described on this reference page is aligned with the ISO C standard. Any
7948 conflict between the requirements described here and the ISO C standard is unintentional. This
7949 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7950 These functions shall compute the complex sine of *z*.

7951 RETURN VALUE

7952 These functions shall return the complex sine value.

7953 ERRORS

7954 No errors are defined.

7955 EXAMPLES

7956 None.

7957 APPLICATION USAGE

7958 None.

7959 RATIONALE

7960 None.

7961 FUTURE DIRECTIONS

7962 None.

7963 SEE ALSO

7964 *casin()*, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

7965 CHANGE HISTORY

7966 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

7967 NAME

7968 **cosh, coshf, coshl** — complex hyperbolic cosine functions

7969 SYNOPSIS

```
7970       #include <complex.h>
7971       double complex cosh(double complex z);
7972       float complex coshf(float complex z);
7973       long double complex coshl(long double complex z);
```

7974 DESCRIPTION

7975 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
7976 conflict between the requirements described here and the ISO C standard is unintentional. This
7977 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

7978 These functions shall compute the complex hyperbolic cosine of *z*.

7979 RETURN VALUE

7980 These functions shall return the complex hyperbolic cosine value.

7981 ERRORS

7982 No errors are defined.

7983 EXAMPLES

7984 None.

7985 APPLICATION USAGE

7986 None.

7987 RATIONALE

7988 None.

7989 FUTURE DIRECTIONS

7990 None.

7991 SEE ALSO

7992 **cosh()**, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

7993 CHANGE HISTORY

7994 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

```
7995 NAME
7996      csinl — complex sine functions
7997 SYNOPSIS
7998      #include <complex.h>
7999      long double complex csinl(long double complex z);
8000 DESCRIPTION
8001      Refer to csin().
```

8002 NAME

8003 `csqrt`, `csqrft`, `csqrtl` — complex square root functions

8004 SYNOPSIS

```
8005        #include <complex.h>
8006
8007        double complex csqrt(double complex z);
8008        float complex csqrft(float complex z);
8009        long double complex csqrtl(long double complex z);
```

8009 DESCRIPTION

8010 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8011 conflict between the requirements described here and the ISO C standard is unintentional. This
8012 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

8013 These functions shall compute the complex square root of *z*, with a branch cut along the
8014 negative real axis.

8015 RETURN VALUE

8016 These functions shall return the complex square root value, in the range of the right half-plane
8017 (including the imaginary axis).

8018 ERRORS

8019 No errors are defined.

8020 EXAMPLES

8021 None.

8022 APPLICATION USAGE

8023 None.

8024 RATIONALE

8025 None.

8026 FUTURE DIRECTIONS

8027 None.

8028 SEE ALSO

8029 `cabs()`, `cpow()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<complex.h>`

8030 CHANGE HISTORY

8031 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

8032 NAME

8033 `ctan`, `ctanf`, `ctanl` — complex tangent functions

8034 SYNOPSIS

```
8035        #include <complex.h>
8036
8037        double complex ctan(double complex z);
8038        float complex ctanf(float complex z);
8039        long double complex ctanl(long double complex z);
```

8039 DESCRIPTION

8040 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8041 conflict between the requirements described here and the ISO C standard is unintentional. This
8042 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

8043 These functions shall compute the complex tangent of *z*.

8044 RETURN VALUE

8045 These functions shall return the complex tangent value.

8046 ERRORS

8047 No errors are defined.

8048 EXAMPLES

8049 None.

8050 APPLICATION USAGE

8051 None.

8052 RATIONALE

8053 None.

8054 FUTURE DIRECTIONS

8055 None.

8056 SEE ALSO

8057 `catan()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<complex.h>`

8058 CHANGE HISTORY

8059 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

8060 NAME

8061 `ctanh`, `ctanhf`, `ctanhl` — complex hyperbolic tangent functions

8062 SYNOPSIS

```
8063     #include <complex.h>
8064
8065     double complex ctanh(double complex z);
8066     float complex ctanhf(float complex z);
8067     long double complex ctanhl(long double complex z);
```

8068 DESCRIPTION

8069 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8070 conflict between the requirements described here and the ISO C standard is unintentional. This
8071 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

8072 These functions shall compute the complex hyperbolic tangent of *z*.

8073 RETURN VALUE

8074 These functions shall return the complex hyperbolic tangent value.

8075 ERRORS

8076 No errors are defined.

8077 EXAMPLES

8078 None.

8079 APPLICATION USAGE

8080 None.

8081 RATIONALE

8082 None.

8083 FUTURE DIRECTIONS

8084 None.

8085 SEE ALSO

8086 `catanh()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<complex.h>`

8087 CHANGE HISTORY

8088 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

```
8088 NAME
8089      ctanl — complex tangent functions
8090 SYNOPSIS
8091      #include <complex.h>
8092      long double complex ctanl(long double complex z);
8093 DESCRIPTION
8094      Refer to ctan().
```

8095 **NAME**

8096 *ctermid* — generate a pathname for the controlling terminal

8097 **SYNOPSIS**

8098 CX #include <stdio.h>

8099 char *ctermid(char **s*);

8100

8101 **DESCRIPTION**

8102 The *ctermid*() function shall generate a string that, when used as a pathname, refers to the
8103 current controlling terminal for the current process. If *ctermid*() returns a pathname, access to the
8104 file is not guaranteed.

8105 If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS
8106 functions, it shall ensure that the *ctermid*() function is called with a non-NULL parameter.

8107 **RETURN VALUE**

8108 If *s* is a null pointer, the string shall be generated in an area that may be static (and therefore may
8109 be overwritten by each call), the address of which shall be returned. Otherwise, *s* is assumed to
8110 point to a character array of at least L_ctermid bytes; the string is placed in this array and the
8111 value of *s* shall be returned. The symbolic constant L_ctermid is defined in <stdio.h>, and shall
8112 have a value greater than 0.

8113 The *ctermid*() function shall return an empty string if the pathname that would refer to the
8114 controlling terminal cannot be determined, or if the function is unsuccessful.

8115 **ERRORS**

8116 No errors are defined.

8117 **EXAMPLES**8118 **Determining the Controlling Terminal for the Current Process**

8119 The following example returns a pointer to a string that identifies the controlling terminal for the
8120 current process. The pathname for the terminal is stored in the array pointed to by the *ptr*
8121 argument, which has a size of L_ctermid bytes, as indicated by the *term* argument.

```
8122 #include <stdio.h>
8123 ...
8124 char term[L_ctermid];
8125 char *ptr;
8126 ptr = ctermid(term);
```

8127 **APPLICATION USAGE**

8128 The difference between *ctermid*() and *ttynname*() is that *ttynname*() must be handed a file
8129 descriptor and return a path of the terminal associated with that file descriptor, while *ctermid*()
8130 returns a string (such as "/dev/tty") that refers to the current controlling terminal if used as a
8131 pathname.

8132 **RATIONALE**

8133 L_ctermid must be defined appropriately for a given implementation and must be greater than
8134 zero so that array declarations using it are accepted by the compiler. The value includes the
8135 terminating null byte.

8136 Conforming applications that use threads cannot call *ctermid*() with NULL as the parameter if
8137 either _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS is defined. If *s* is not
8138 NULL, the *ctermid*() function generates a string that, when used as a pathname, refers to the

8139 current controlling terminal for the current process. If *s* is NULL, the return value of *ctermid()* is
8140 undefined.

8141 There is no additional burden on the programmer—changing to use a hypothetical thread-safe
8142 version of *ctermid()* along with allocating a buffer is more of a burden than merely allocating a
8143 buffer. Application code should not assume that the returned string is short, as some
8144 implementations have more than two pathname components before reaching a logical device
8145 name.

8146 **FUTURE DIRECTIONS**

8147 None.

8148 **SEE ALSO**

8149 *ttyname()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

8150 **CHANGE HISTORY**

8151 First released in Issue 1. Derived from Issue 1 of the SVID.

8152 **Issue 5**

8153 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

8154 **Issue 6**

8155 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

8156 NAME

8157 `ctime`, `ctime_r` — convert a time value to a date and time string

8158 SYNOPSIS

```
8159       #include <time.h>
8160       char *ctime(const time_t *clock);
8161 TSF      char *ctime_r(const time_t *clock, char *buf);
8162
```

8163 DESCRIPTION

8164 CX For `ctime()`: 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.

8167 The `ctime()` function shall convert the time pointed to by `clock`, representing time in seconds since the Epoch, to local time in the form of a string. It shall be equivalent to:

```
8169       asctime(localtime(clock))
```

8170 CX 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 `char`. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions.

8173 The `ctime()` function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

8175 TSF The `ctime_r()` function shall convert the calendar time pointed to by `clock` to local time in exactly the same form as `ctime()` and put the string into the array pointed to by `buf` (which shall be at least 26 bytes in size) and return `buf`.

8178 Unlike `ctime()`, the thread-safe version `ctime_r()` is not required to set `tzname`.

8179 RETURN VALUE

8180 The `ctime()` function shall return the pointer returned by `asctime()` with that broken-down time as an argument.

8182 TSF Upon successful completion, `ctime_r()` shall return a pointer to the string pointed to by `buf`. When an error is encountered, a null pointer shall be returned.

8184 ERRORS

8185 No errors are defined.

8186 EXAMPLES

8187 None.

8188 APPLICATION USAGE

8189 Values for the broken-down time structure can be obtained by calling `gmtime()` or `localtime()`. The `ctime()` function is included for compatibility with older implementations, and does not support localized date and time formats. Applications should use the `strftime()` function to achieve maximum portability.

8193 The `ctime_r()` function is thread-safe and shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

8195 RATIONALE

8196 None.

8197 FUTURE DIRECTIONS

8198 None.

8199 SEE ALSO

8200 *asctime()*, *clock()*, *difftime()*, *gmtime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
8201 the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

8202 CHANGE HISTORY

8203 First released in Issue 1. Derived from Issue 1 of the SVID.

8204 Issue 5

8205 Normative text previously in the APPLICATION USAGE section is moved to the
8206 DESCRIPTION.

8207 The *ctime_r()* function is included for alignment with the POSIX Threads Extension.

8208 A note indicating that the *ctime()* function need not be reentrant is added to the DESCRIPTION.

8209 Issue 6

8210 Extensions beyond the ISO C standard are marked.

8211 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

8212 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
8213 its avoidance of possibly using a static data area.

8214 **NAME**

8215 daylight — daylight savings time flag

8216 **SYNOPSIS**

8217 XSI #include <time.h>

8218 extern int daylight;

8219

8220 **DESCRIPTION**8221 Refer to *tzset()*.

8222 NAME

8223 dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, dbm_nextkey,
 8224 dbm_open, dbm_store — database functions

8225 SYNOPSIS

8226 XSI

```
#include <ndbm.h>

8227     int dbm_clearerr(DBM *db);
8228     void dbm_close(DBM *db);
8229     int dbm_delete(DBM *db, datum key);
8230     int dbm_error(DBM *db);
8231     datum dbm_fetch(DBM *db, datum key);
8232     datum dbm_firstkey(DBM *db);
8233     datum dbm_nextkey(DBM *db);
8234     DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);
8235     int dbm_store(DBM *db, datum key, datum content, int store_mode);
```

8237 DESCRIPTION

8238 These functions create, access, and modify a database.

8239 A **datum** consists of at least two members, *dptr* and *dsiz*. The *dptr* member points to an object
 8240 that is *dsiz* bytes in length. Arbitrary binary data, as well as character strings, may be stored in
 8241 the object pointed to by *dptr*.

8242 The database is stored in two files. One file is a directory containing a bitmap of keys and has
 8243 .dir as its suffix. The second file contains all data and has .pag as its suffix.

8244 The *dbm_open()* function shall open a database. The *file* argument to the function is the
 8245 pathname of the database. The function opens two files named *file.dir* and *file.pag*. The
 8246 *open_flags* argument has the same meaning as the *flags* argument of *open()* except that a database
 8247 opened for write-only access opens the files for read and write access and the behavior of the
 8248 O_APPEND flag is unspecified. The *file_mode* argument has the same meaning as the third
 8249 argument of *open()*.

8250 The *dbm_close()* function shall close a database. The application shall ensure that argument *db* is
 8251 a pointer to a **dbm** structure that has been returned from a call to *dbm_open()*.

8252 These database functions shall support an internal block size large enough to support
 8253 key/content pairs of at least 1 023 bytes.

8254 The *dbm_fetch()* function shall read a record from a database. The argument *db* is a pointer to a
 8255 database structure that has been returned from a call to *dbm_open()*. The argument *key* is a
 8256 **datum** that has been initialized by the application to the value of the key that matches the key of
 8257 the record the program is fetching.

8258 The *dbm_store()* function shall write a record to a database. The argument *db* is a pointer to a
 8259 database structure that has been returned from a call to *dbm_open()*. The argument *key* is a
 8260 **datum** that has been initialized by the application to the value of the key that identifies (for
 8261 subsequent reading, writing, or deleting) the record the application is writing. The argument
 8262 *content* is a **datum** that has been initialized by the application to the value of the record the
 8263 program is writing. The argument *store_mode* controls whether *dbm_store()* replaces any pre-
 8264 existing record that has the same key that is specified by the *key* argument. The application shall
 8265 set *store_mode* to either DBM_INSERT or DBM_REPLACE. If the database contains a record that
 8266 matches the *key* argument and *store_mode* is DBM_REPLACE, the existing record shall be
 8267 replaced with the new record. If the database contains a record that matches the *key* argument
 8268 and *store_mode* is DBM_INSERT, the existing record shall be left unchanged and the new record

8269 ignored. If the database does not contain a record that matches the *key* argument and *store_mode*
8270 is either DBM_INSERT or DBM_REPLACE, the new record shall be inserted in the database.

8271 If the sum of a key/content pair exceeds the internal block size, the result is unspecified.
8272 Moreover, the application shall ensure that all key/content pairs that hash together fit on a
8273 single block. The *dbm_store()* function shall return an error in the event that a disk block fills
8274 with inseparable data.

8275 The *dbm_delete()* function shall delete a record and its key from the database. The argument *db* is
8276 a pointer to a database structure that has been returned from a call to *dbm_open()*. The argument
8277 *key* is a **datum** that has been initialized by the application to the value of the key that identifies
8278 the record the program is deleting.

8279 The *dbm_firstkey()* function shall return the first key in the database. The argument *db* is a
8280 pointer to a database structure that has been returned from a call to *dbm_open()*.

8281 The *dbm_nextkey()* function shall return the next key in the database. The argument *db* is a
8282 pointer to a database structure that has been returned from a call to *dbm_open()*. The application
8283 shall ensure that the *dbm_firstkey()* function is called before calling *dbm_nextkey()*. Subsequent
8284 calls to *dbm_nextkey()* return the next key until all of the keys in the database have been
8285 returned.

8286 The *dbm_error()* function shall return the error condition of the database. The argument *db* is a
8287 pointer to a database structure that has been returned from a call to *dbm_open()*.

8288 The *dbm_clearerr()* function shall clear the error condition of the database. The argument *db* is a
8289 pointer to a database structure that has been returned from a call to *dbm_open()*.

8290 The *dptr* pointers returned by these functions may point into static storage that may be changed
8291 by subsequent calls.

8292 These functions need not be reentrant. A function that is not required to be reentrant is not
8293 required to be thread-safe.

8294 RETURN VALUE

8295 The *dbm_store()* and *dbm_delete()* functions shall return 0 when they succeed and a negative
8296 value when they fail.

8297 The *dbm_store()* function shall return 1 if it is called with a *flags* value of DBM_INSERT and the
8298 function finds an existing record with the same key.

8299 The *dbm_error()* function shall return 0 if the error condition is not set and return a non-zero
8300 value if the error condition is set.

8301 The return value of *dbm_clearerr()* is unspecified.

8302 The *dbm_firstkey()* and *dbm_nextkey()* functions shall return a key **datum**. When the end of the
8303 database is reached, the *dptr* member of the key is a null pointer. If an error is detected, the *dptr*
8304 member of the key shall be a null pointer and the error condition of the database shall be set.

8305 The *dbm_fetch()* function shall return a content **datum**. If no record in the database matches the
8306 key or if an error condition has been detected in the database, the *dptr* member of the content
8307 shall be a null pointer.

8308 The *dbm_open()* function shall return a pointer to a database structure. If an error is detected
8309 during the operation, *dbm_open()* shall return a (**DBM ***)0.

8310 ERRORS

8311 No errors are defined.

8312 EXAMPLES

8313 None.

8314 APPLICATION USAGE

8315 The following code can be used to traverse the database:

```
8316     for(key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

8317 The *dbm_** functions provided in this library should not be confused in any way with those of a
8318 general-purpose database management system. These functions do not provide for multiple
8319 search keys per entry, they do not protect against multi-user access (in other words they do not
8320 lock records or files), and they do not provide the many other useful database functions that are
8321 found in more robust database management systems. Creating and updating databases by use of
8322 these functions is relatively slow because of data copies that occur upon hash collisions. These
8323 functions are useful for applications requiring fast lookup of relatively static information that is
8324 to be indexed by a single key.

8325 Note that a strictly conforming application is extremely limited by these functions: since there is
8326 no way to determine that the keys in use do not all hash to the same value (although that would
8327 be rare), a strictly conforming application cannot be guaranteed that it can store more than one
8328 block's worth of data in the database. As long as a key collision does not occur, additional data
8329 may be stored, but because there is no way to determine whether an error is due to a key
8330 collision or some other error condition (*dbm_error()* being effectively a Boolean), once an error is
8331 detected, the application is effectively limited to guessing what the error might be if it wishes to
8332 continue using these functions.

8333 The *dbm_delete()* function need not physically reclaim file space, although it does make it
8334 available for reuse by the database.

8335 After calling *dbm_store()* or *dbm_delete()* during a pass through the keys by *dbm_firstkey()* and
8336 *dbm_nextkey()*, the application should reset the database by calling *dbm_firstkey()* before again
8337 calling *dbm_nextkey()*. The contents of these files are unspecified and may not be portable.

8338 RATIONALE

8339 None.

8340 FUTURE DIRECTIONS

8341 None.

8342 SEE ALSO

8343 *open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <ndbm.h>

8344 CHANGE HISTORY

8345 First released in Issue 4, Version 2.

8346 Issue 5

8347 Moved from X/OPEN UNIX extension to BASE.

8348 Normative text previously in the APPLICATION USAGE section is moved to the
8349 DESCRIPTION.

8350 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

8351 **Issue 6**

8352 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

8353 NAME

8354 **difftime** — compute the difference between two calendar time values

8355 SYNOPSIS

8356 **#include <time.h>**

8357 **double difftime(time_t time1, time_t time0);**

8358 DESCRIPTION

8359 **CX** The functionality described on this reference page is aligned with the ISO C standard. Any
8360 conflict between the requirements described here and the ISO C standard is unintentional. This
8361 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

8362 The *difftime()* function shall compute the difference between two calendar times (as returned by
8363 *time()*): *time1*–*time0*.

8364 RETURN VALUE

8365 The *difftime()* function shall return the difference expressed in seconds as a type **double**.

8366 ERRORS

8367 No errors are defined.

8368 EXAMPLES

8369 None.

8370 APPLICATION USAGE

8371 None.

8372 RATIONALE

8373 None.

8374 FUTURE DIRECTIONS

8375 None.

8376 SEE ALSO

8377 *asctime()*, *clock()*, *ctime()*, *gmtime()*, *localtime()*, *mkttime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
8378 the Base Definitions volume of IEEE Std 1003.1-2001, **<time.h>**

8379 CHANGE HISTORY

8380 First released in Issue 4. Derived from the ISO C standard.

8381 **NAME**

8382 dirname — report the parent directory name of a file pathname

8383 **SYNOPSIS**

8384 XSI #include <libgen.h>

8385 char *dirname(char *path);

8386

8387 **DESCRIPTION**

8388 The *dirname()* function shall take a pointer to a character string that contains a pathname, and
 8389 return a pointer to a string that is a pathname of the parent directory of that file. Trailing '/'
 8390 characters in the path are not counted as part of the path.

8391 If *path* does not contain a '/', then *dirname()* shall return a pointer to the string ". ". If *path* is a
 8392 null pointer or points to an empty string, *dirname()* shall return a pointer to the string ". ".

8393 The *dirname()* function need not be reentrant. A function that is not required to be reentrant is
 8394 not required to be thread-safe.

8395 **RETURN VALUE**

8396 The *dirname()* function shall return a pointer to a string that is the parent directory of *path*. If
 8397 *path* is a null pointer or points to an empty string, a pointer to a string ". " is returned.

8398 The *dirname()* function may modify the string pointed to by *path*, and may return a pointer to
 8399 static storage that may then be overwritten by subsequent calls to *dirname()*.

8400 **ERRORS**

8401 No errors are defined.

8402 **EXAMPLES**

8403 The following code fragment reads a pathname, changes the current working directory to the
 8404 parent directory, and opens the file.

```
8405 char path[PATH_MAX], *pathcopy;
8406 int fd;
8407 fgets(path, PATH_MAX, stdin);
8408 pathcopy = strdup(path);
8409 chdir(dirname(pathcopy));
8410 fd = open(basename(path), O_RDONLY);
```

8411 **Sample Input and Output Strings for dirname()**

8412 In the following table, the input string is the value pointed to by *path*, and the output string is
 8413 the return value of the *dirname()* function.

8414	Input String	Output String
8415	"/usr/lib"	"/usr"
8416	"/usr/"	"/"
8417	"usr"	". "
8418	"/"	"/"
8419	". "	". "
8420	". ."	". ."

8421 Changing the Current Directory to the Parent Directory

8422 The following program fragment reads a pathname, changes the current working directory to
8423 the parent directory, and opens the file.

```
8424     #include <unistd.h>
8425     #include <limits.h>
8426     #include <stdio.h>
8427     #include <fcntl.h>
8428     #include <string.h>
8429     #include <libgen.h>
8430     ...
8431     char path[PATH_MAX], *pathcopy;
8432     int fd;
8433     ...
8434     fgets(path, PATH_MAX, stdin);
8435     pathcopy = strdup(path);
8436     chdir(dirname(pathcopy));
8437     fd = open(basename(path), O_RDONLY);
```

8438 APPLICATION USAGE

8439 The *dirname()* and *basename()* functions together yield a complete pathname. The expression
8440 *dirname(path)* obtains the pathname of the directory where *basename(path)* is found.

8441 Since the meaning of the leading " / " is implementation-defined, *dirname("//foo")* may return
8442 either " / " or ' / ' (but nothing else).

8443 RATIONALE

8444 None.

8445 FUTURE DIRECTIONS

8446 None.

8447 SEE ALSO

8448 *basename()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**libgen.h**>

8449 CHANGE HISTORY

8450 First released in Issue 4, Version 2.

8451 Issue 5

8452 Moved from X/OPEN UNIX extension to BASE.

8453 Normative text previously in the APPLICATION USAGE section is moved to the
8454 DESCRIPTION.

8455 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

8456 NAME

8457

— compute the quotient and remainder of an integer division

8458 SYNOPSIS

8459

```
#include <stdlib.h>
```


8460

```
div_t div(int numer, int denom);
```

8461 DESCRIPTION

8462 CX The functionality described on this reference page is aligned with the ISO C standard. Any
8463 conflict between the requirements described here and the ISO C standard is unintentional. This
8464 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

8465 The *div()* function shall compute the quotient and remainder of the division of the numerator
8466 *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer
8467 of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be
8468 represented, the behavior is undefined; otherwise, *quot***denom*+*rem* shall equal *numer*.

8469 RETURN VALUE

8470 The *div()* function shall return a structure of type **div_t**, comprising both the quotient and the
8471 remainder. The structure includes the following members, in any order:

8472

```
int quot; /* quotient */
```


8473

```
int rem; /* remainder */
```

8474 ERRORS

8475 No errors are defined.

8476 EXAMPLES

8477 None.

8478 APPLICATION USAGE

8479 None.

8480 RATIONALE

8481 None.

8482 FUTURE DIRECTIONS

8483 None.

8484 SEE ALSO

8485 *ldiv()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

8486 CHANGE HISTORY

8487 First released in Issue 4. Derived from the ISO C standard.

8488 NAME

8489 *dlclose* — close a *dlopen()* object

8490 SYNOPSIS

8491 XSI #include <dlfcn.h>

```
8492        int dlclose(void *handle);
```

8493

8494 DESCRIPTION

8495 The *dlclose()* function shall inform the system that the object referenced by a *handle* returned
8496 from a previous *dlopen()* invocation is no longer needed by the application.

8497 The use of *dlclose()* reflects a statement of intent on the part of the process, but does not create
8498 any requirement upon the implementation, such as removal of the code or symbols referenced
8499 by *handle*. Once an object has been closed using *dlclose()* an application should assume that its
8500 symbols are no longer available to *dlsym()*. All objects loaded automatically as a result of
8501 invoking *dlopen()* on the referenced object shall also be closed if this is the last reference to it.

8502 Although a *dlclose()* operation is not required to remove structures from an address space,
8503 neither is an implementation prohibited from doing so. The only restriction on such a removal is
8504 that no object shall be removed to which references have been relocated, until or unless all such
8505 references are removed. For instance, an object that had been loaded with a *dlopen()* operation
8506 specifying the RTLD_GLOBAL flag might provide a target for dynamic relocations performed in
8507 the processing of other objects—in such environments, an application may assume that no
8508 relocation, once made, shall be undone or remade unless the object requiring the relocation has
8509 itself been removed.

8510 RETURN VALUE

8511 If the referenced object was successfully closed, *dlclose()* shall return 0. If the object could not be
8512 closed, or if *handle* does not refer to an open object, *dlclose()* shall return a non-zero value. More
8513 detailed diagnostic information shall be available through *dlerror()*.

8514 ERRORS

8515 No errors are defined.

8516 EXAMPLES

8517 The following example illustrates use of *dlopen()* and *dlclose()*:

```
8518        ...
8519        /* Open a dynamic library and then close it ... */
8520        #include <dlfcn.h>
8521        void *mylib;
8522        int eret;
8523
8524        mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
8525        ...
8526        eret = dlclose(mylib);
8527        ...
```

8527 APPLICATION USAGE

8528 A conforming application should employ a *handle* returned from a *dlopen()* invocation only
8529 within a given scope bracketed by the *dlopen()* and *dlclose()* operations. Implementations are
8530 free to use reference counting or other techniques such that multiple calls to *dlopen()* referencing
8531 the same object may return the same object for *handle*. Implementations are also free to reuse a
8532 *handle*. For these reasons, the value of a *handle* must be treated as an opaque object by the
8533 application, used only in calls to *dlsym()* and *dlclose()*.

8534 RATIONALE

8535 None.

8536 FUTURE DIRECTIONS

8537 None.

8538 SEE ALSO

8539 *dlerror()*, *dlopen()*, *dlsym()*, the Base Definitions volume of IEEE Std 1003.1-2001, <dlfcn.h>

8540 CHANGE HISTORY

8541 First released in Issue 5.

8542 Issue 6

8543 The DESCRIPTION is updated to say that the referenced object is closed “if this is the last reference to it”.

8545 NAME

8546 *dlerror* — get diagnostic information

8547 SYNOPSIS

8548 XSI #include <dlfcn.h>
8549
8550 char *dlerror(void);

8551 DESCRIPTION

8552 The *dlerror()* function shall return a null-terminated character string (with no trailing <newline>) that describes the last error that occurred during dynamic linking processing. If no dynamic linking errors have occurred since the last invocation of *dlerror()*, *dlerror()* shall return NULL. Thus, invoking *dlerror()* a second time, immediately following a prior invocation, shall result in NULL being returned.

8557 The *dlerror()* function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

8559 RETURN VALUE

8560 If successful, *dlerror()* shall return a null-terminated character string; otherwise, NULL shall be returned.

8562 ERRORS

8563 No errors are defined.

8564 EXAMPLES

8565 The following example prints out the last dynamic linking error:

```
8566        ...
8567        #include <dlfcn.h>
8568        char *errstr;
8569        errstr = dlerror();
8570        if (errstr != NULL)
8571        printf ("A dynamic linking error occurred: (%s)\n", errstr);
8572        ...
```

8573 APPLICATION USAGE

8574 The messages returned by *dlerror()* may reside in a static buffer that is overwritten on each call to *dlerror()*. Application code should not write to this buffer. Programs wishing to preserve an error message should make their own copies of that message. Depending on the application environment with respect to asynchronous execution events, such as signals or other asynchronous computation sharing the address space, conforming applications should use a critical section to retrieve the error pointer and buffer.

8580 RATIONALE

8581 None.

8582 FUTURE DIRECTIONS

8583 None.

8584 SEE ALSO

8585 *dlclose()*, *dlopen()*, *dlsym()*, the Base Definitions volume of IEEE Std 1003.1-2001, <dlfcn.h>

8586 **CHANGE HISTORY**

8587 First released in Issue 5.

8588 **Issue 6**

8589 In the DESCRIPTION the note about reentrancy and thread-safety is added.

8590 **NAME**

8591 *dlopen* — gain access to an executable object file

8592 **SYNOPSIS**

8593 XSI #include <dlfcn.h>

8594 void *dlopen(const char *file, int mode);

8595

8596 **DESCRIPTION**

8597 The *dlopen*() function shall make an executable object file specified by *file* available to the calling
8598 program. The class of files eligible for this operation and the manner of their construction are
8599 implementation-defined, though typically such files are executable objects such as shared
8600 libraries, relocatable files, or programs. Note that some implementations permit the construction
8601 of dependencies between such objects that are embedded within files. In such cases, a *dlopen*()
8602 operation shall load such dependencies in addition to the object referenced by *file*.
8603 Implementations may also impose specific constraints on the construction of programs that can
8604 employ *dlopen*() and its related services.

8605 A successful *dlopen*() shall return a *handle* which the caller may use on subsequent calls to
8606 *dlsym*() and *dlclose*(). The value of this *handle* should not be interpreted in any way by the caller.

8607 The *file* argument is used to construct a pathname to the object file. If *file* contains a slash
8608 character, the *file* argument is used as the pathname for the file. Otherwise, *file* is used in an
8609 implementation-defined manner to yield a pathname.

8610 If the value of *file* is 0, *dlopen*() shall provide a *handle* on a global symbol object. This object shall
8611 provide access to the symbols from an ordered set of objects consisting of the original program
8612 image file, together with any objects loaded at program start-up as specified by that process
8613 image file (for example, shared libraries), and the set of objects loaded using a *dlopen*() operation
8614 together with the RTLD_GLOBAL flag. As the latter set of objects can change during execution,
8615 the set identified by *handle* can also change dynamically.

8616 Only a single copy of an object file is brought into the address space, even if *dlopen*() is invoked
8617 multiple times in reference to the file, and even if different pathnames are used to reference the
8618 file.

8619 The *mode* parameter describes how *dlopen*() shall operate upon *file* with respect to the processing
8620 of relocations and the scope of visibility of the symbols provided within *file*. When an object is
8621 brought into the address space of a process, it may contain references to symbols whose
8622 addresses are not known until the object is loaded. These references shall be relocated before the
8623 symbols can be accessed. The *mode* parameter governs when these relocations take place and
8624 may have the following values:

8625 RTLD_LAZY Relocations shall be performed at an implementation-defined time,
8626 ranging from the time of the *dlopen*() call until the first reference to a
8627 given symbol occurs. Specifying RTLD_LAZY should improve
8628 performance on implementations supporting dynamic symbol binding as
8629 a process may not reference all of the functions in any given object. And,
8630 for systems supporting dynamic symbol resolution for normal process
8631 execution, this behavior mimics the normal handling of process
8632 execution.

8633 RTLD_NOW All necessary relocations shall be performed when the object is first
8634 loaded. This may waste some processing if relocations are performed for
8635 functions that are never referenced. This behavior may be useful for
8636 applications that need to know as soon as an object is loaded that all

8637		symbols referenced during execution are available.	
8638		Any object loaded by <i>dlopen()</i> that requires relocations against global symbols can reference the symbols in the original process image file, any objects loaded at program start-up, from the object itself as well as any other object included in the same <i>dlopen()</i> invocation, and any objects that were loaded in any <i>dlopen()</i> invocation and which specified the RTLD_GLOBAL flag. To determine the scope of visibility for the symbols loaded with a <i>dlopen()</i> invocation, the <i>mode</i> parameter should be a bitwise-inclusive OR with one of the following values:	
8644	RTLD_GLOBAL	The object's symbols shall be made available for the relocation processing of any other object. In addition, symbol lookup using <i>dlopen(0, mode)</i> and an associated <i>dlsym()</i> allows objects loaded with this <i>mode</i> to be searched.	
8647	RTLD_LOCAL	The object's symbols shall not be made available for the relocation processing of any other object.	
8649	If neither RTLD_GLOBAL nor RTLD_LOCAL are specified, then the default behavior is		2
8650	unspecified.		2
8651	If a <i>file</i> is specified in multiple <i>dlopen()</i> invocations, <i>mode</i> is interpreted at each invocation. Note, however, that once RTLD_NOW has been specified all relocations shall have been completed rendering further RTLD_NOW operations redundant and any further RTLD_LAZY operations irrelevant. Similarly, note that once RTLD_GLOBAL has been specified the object shall maintain the RTLD_GLOBAL status regardless of any previous or future specification of RTLD_LOCAL, as long as the object remains in the address space (see <i>dlclose()</i>).		
8657	Symbols introduced into a program through calls to <i>dlopen()</i> may be used in relocation activities. Symbols so introduced may duplicate symbols already defined by the program or previous <i>dlopen()</i> operations. To resolve the ambiguities such a situation might present, the resolution of a symbol reference to symbol definition is based on a symbol resolution order. Two such resolution orders are defined: <i>load</i> or <i>dependency</i> ordering. Load order establishes an ordering among symbol definitions, such that the definition first loaded (including definitions from the image file and any dependent objects loaded with it) has priority over objects added later (via <i>dlopen()</i>). Load ordering is used in relocation processing. Dependency ordering uses a breadth-first order starting with a given object, then all of its dependencies, then any dependents of those, iterating until all dependencies are satisfied. With the exception of the global symbol object obtained via a <i>dlopen()</i> operation on a <i>file</i> of 0, dependency ordering is used by the <i>dlsym()</i> function. Load ordering is used in <i>dlsym()</i> operations upon the global symbol object.		
8669	When an object is first made accessible via <i>dlopen()</i> it and its dependent objects are added in dependency order. Once all the objects are added, relocations are performed using load order. Note that if an object or its dependencies had been previously loaded, the load and dependency orders may yield different resolutions.		
8673	The symbols introduced by <i>dlopen()</i> operations and available through <i>dlsym()</i> are at a minimum those which are exported as symbols of global scope by the object. Typically such symbols shall be those that were specified in (for example) C source code as having <i>extern</i> linkage. The precise manner in which an implementation constructs the set of exported symbols for a <i>dlopen()</i> object is specified by that implementation.		
8678	RETURN VALUE		
8679	If <i>file</i> cannot be found, cannot be opened for reading, is not of an appropriate object format for processing by <i>dlopen()</i> , or if an error occurs during the process of loading <i>file</i> or relocating its symbolic references, <i>dlopen()</i> shall return NULL. More detailed diagnostic information shall be available through <i>dlerror()</i> .		

8683 ERRORS

8684 No errors are defined.

8685 EXAMPLES

8686 None.

8687 APPLICATION USAGE

8688 None.

8689 RATIONALE

8690 None.

8691 FUTURE DIRECTIONS

8692 None.

8693 SEE ALSO

8694 *dlclose()*, *dlerror()*, *dlsym()*, the Base Definitions volume of IEEE Std 1003.1-2001, <dlfcn.h>

8695 CHANGE HISTORY

8696 First released in Issue 5.

8697 Issue 6

8698 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/21 is applied, changing the default 2
8699 behavior in the DESCRIPTION when neither RTLD_GLOBAL nor RTLD_LOCAL are specified 2
8700 from implementation-defined to unspecified. 2

8701 **NAME**

8702 *dlsym* — obtain the address of a symbol from a *dlopen()* object

8703 **SYNOPSIS**

8704 XSI #include <dlfcn.h>

```
8705        void *dlsym(void *restrict handle, const char *restrict name);
```

8706

8707 **DESCRIPTION**

8708 The *dlsym()* function shall obtain the address of a symbol defined within an object made
 8709 accessible through a *dlopen()* call. The *handle* argument is the value returned from a call to
 8710 *dlopen()* (and which has not since been released via a call to *dlclose()*), and *name* is the symbol's
 8711 name as a character string.

8712 The *dlsym()* function shall search for the named symbol in all objects loaded automatically as a
 8713 result of loading the object referenced by *handle* (see *dlopen()*). Load ordering is used in *dlsym()*
 8714 operations upon the global symbol object. The symbol resolution algorithm used shall be
 8715 dependency order as described in *dlopen()*.

8716 The RTLD_DEFAULT and RTLD_NEXT flags are reserved for future use.

8717 **RETURN VALUE**

8718 If *handle* does not refer to a valid object opened by *dlopen()*, or if the named symbol cannot be
 8719 found within any of the objects associated with *handle*, *dlsym()* shall return NULL. More
 8720 detailed diagnostic information shall be available through *dlerror()*.

8721 **ERRORS**

8722 No errors are defined.

8723 **EXAMPLES**

8724 The following example shows how *dlopen()* and *dlsym()* can be used to access either function or
 8725 data objects. For simplicity, error checking has been omitted.

```
8726        void        *handle;
8727        int        *iptr, (*fptr)(int);
8728        /* open the needed object */
8729        handle = dlopen("/usr/home/me/libfoo.so", RTLD_LOCAL | RTLD_LAZY);
8730        /* find the address of function and data objects */
8731        *(void **)(&fptr) = dlsym(handle, "my_function");
8732        iptr = (int *)dlsym(handle, "my_object");
8733        /* invoke function, passing value of integer as a parameter */
8734        (*fptr)(*iptr);
```

1
1

8735 **APPLICATION USAGE**

8736 Special purpose values for *handle* are reserved for future use. These values and their meanings
 8737 are:

8738 RTLD_DEFAULT The symbol lookup happens in the normal global scope; that is, a search for a
 8739 symbol using this handle would find the same definition as a direct use of this
 8740 symbol in the program code.

8741 RTLD_NEXT Specifies the next object after this one that defines *name*. This *one* refers to the
 8742 object containing the invocation of *dlsym()*. The *next* object is the one found
 8743 upon the application of a load order symbol resolution algorithm (see
 8744 *dlopen()*). The next object is either one of global scope (because it was
 8745 introduced as part of the original process image or because it was added with

8746 a *dlopen()* operation including the RTLD_GLOBAL flag), or is an object that
8747 was included in the same *dlopen()* operation that loaded this one.

8748 The RTLD_NEXT flag is useful to navigate an intentionally created hierarchy
8749 of multiply-defined symbols created through *interposition*. For example, if a
8750 program wished to create an implementation of *malloc()* that embedded some
8751 statistics gathering about memory allocations, such an implementation could
8752 use the real *malloc()* definition to perform the memory allocation—and itself
8753 only embed the necessary logic to implement the statistics gathering function.

8754 RATIONALE

8755 The ISO C standard does not require that pointers to functions can be cast back and forth to
8756 pointers to data. Indeed, the ISO C standard does not require that an object of type **void *** can
8757 hold a pointer to a function. Implementations supporting the XSI extension, however, do require
8758 that an object of type **void *** can hold a pointer to a function. The result of converting a pointer to
8759 a function into a pointer to another data type (except **void ***) is still undefined, however. Note
8760 that compilers conforming to the ISO C standard are required to generate a warning if a
8761 conversion from a **void *** pointer to a function pointer is attempted as in:

8762 `fptr = (int (*)(int))dlsym(handle, "my_function");`

8763 Due to the problem noted here, a future version may either add a new function to return
8764 function pointers, or the current interface may be deprecated in favor of two new functions: one
8765 that returns data pointers and the other that returns function pointers.

8766 FUTURE DIRECTIONS

8767 None.

8768 SEE ALSO

8769 *dlclose()*, *dlerror()*, *dlopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, <dlfcn.h>

8770 CHANGE HISTORY

8771 First released in Issue 5.

8772 Issue 6

8773 The **restrict** keyword is added to the *dlsym()* prototype for alignment with the
8774 ISO/IEC 9899:1999 standard.

8775 The RTLD_DEFAULT and RTLD_NEXT flags are reserved for future use.

8776 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/14 is applied, correcting an example, and
8777 adding text to the RATIONALE describing issues related to conversion of pointers to functions
8778 and back again.

8779 NAME

8780 drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, srand48 — generate
 8781 uniformly distributed pseudo-random numbers

8782 SYNOPSIS

```
8783 XSI #include <stdlib.h>
8784
8785     double drand48(void);
8786     double erand48(unsigned short xsubi[3]);
8787     long jrand48(unsigned short xsubi[3]);
8788     void lcong48(unsigned short param[7]);
8789     long lrand48(void);
8790     long mrand48(void);
8791     long nrand48(unsigned short xsubi[3]);
8792     unsigned short *seed48(unsigned short seed16v[3]);
8793     void srand48(long seedval);
```

8794 DESCRIPTION

8795 This family of functions shall generate pseudo-random numbers using a linear congruential
 8796 algorithm and 48-bit integer arithmetic.

8797 The *drand48()* and *erand48()* functions shall return non-negative, double-precision, floating-
 8798 point values, uniformly distributed over the interval [0.0,1.0].

8799 The *lrand48()* and *nrand48()* functions shall return non-negative, long integers, uniformly
 8800 distributed over the interval [0,2³¹).

8801 The *mrand48()* and *jrand48()* functions shall return signed long integers uniformly distributed
 8802 over the interval [-2³¹,2³¹).

8803 The *srand48()*, *seed48()*, and *lcong48()* functions are initialization entry points, one of which
 8804 should be invoked before either *drand48()*, *lrand48()*, or *mrand48()* is called. (Although it is not
 8805 recommended practice, constant default initializer values shall be supplied automatically if
 8806 *drand48()*, *lrand48()*, or *mrand48()* is called without a prior call to an initialization entry point.)
 8807 The *erand48()*, *nrand48()*, and *jrand48()* functions do not require an initialization entry point to
 8808 be called first.

8809 All the routines work by generating a sequence of 48-bit integer values, X_i , according to the
 8810 linear congruential formula:

$$X_{n+1} = (aX_n + c) \bmod m \quad n \geq 0$$

8812 The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48()* is invoked,
 8813 the multiplier value a and the addend value c are given by:

$$a = 5DEECE66D_{16} = 273673163155_8$$

$$c = B_{16} = 13_8$$

8816 The value returned by any of the *drand48()*, *erand48()*, *jrand48()*, *lrand48()*, *mrand48()*, or
 8817 *nrand48()* functions is computed by first generating the next 48-bit X_i in the sequence. Then the
 8818 appropriate number of bits, according to the type of data item to be returned, are copied from
 8819 the high-order (leftmost) bits of X_i and transformed into the returned value.

8820 The *drand48()*, *lrand48()*, and *mrand48()* functions store the last 48-bit X_i generated in an
 8821 internal buffer; that is why the application shall ensure that these are initialized prior to being
 8822 invoked. The *erand48()*, *nrand48()*, and *jrand48()* functions require the calling program to
 8823 provide storage for the successive X_i values in the array specified as an argument when the

functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, *erand48()*, *nrand48()*, and *jrand48()* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers; that is, the sequence of numbers in each stream shall *not* depend upon how many times the routines are called to generate numbers for the other streams.

The initializer function *strand48()* sets the high-order 32 bits of X_i to the low-order 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value 330E₁₆.

The initializer function *seed48()* sets the value of X_i to the 48-bit value specified in the argument array. The low-order 16 bits of X_i are set to the low-order 16 bits of *seed16v[0]*. The mid-order 16 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 low-order 16 bits of *seed16v[2]*. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by *seed48()*, and a pointer to this buffer is the value returned by *seed48()*. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time—use the pointer to get at and store the last X_i value, and then use this value to reinitialize via *seed48()* when the program is restarted.

The initializer function *lcong48()* allows the user to specify the initial X_i , the multiplier value *a*, and the addend value *c*. Argument array elements *param[0-2]* specify X_i , *param[3-5]* specify the multiplier *a*, and *param[6]* specifies the 16-bit addend *c*. After *lcong48()* is called, a subsequent call to either *strand48()* or *seed48()* shall restore the standard multiplier and addend values, *a* and *c*, specified above.

The *drand48()*, *lrand48()*, and *mrnd48()* functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

8847 RETURN VALUE

8848 As described in the DESCRIPTION above.

8849 ERRORS

8850 No errors are defined.

8851 EXAMPLES

8852 None.

8853 APPLICATION USAGE

8854 None.

8855 RATIONALE

8856 None.

8857 FUTURE DIRECTIONS

8858 None.

8859 SEE ALSO

8860 *rand()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

8861 CHANGE HISTORY

8862 First released in Issue 1. Derived from Issue 1 of the SVID.

8863 Issue 5

8864 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

8865 Issue 6

8866 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

8867 **NAME**8868 *dup, dup2* — duplicate an open file descriptor8869 **SYNOPSIS**

```
8870     #include <unistd.h>
8871
8872     int dup(int fildes);
8873     int dup2(int fildes, int fildes2);
```

8873 **DESCRIPTION**

8874 The *dup()* and *dup2()* functions provide an alternative interface to the service provided by
8875 *fcntl()* using the F_DUPFD command. The call:

```
8876     fid = dup(fildes);
```

8877 shall be equivalent to:

```
8878     fid = fcntl(fildes, F_DUPFD, 0);
```

8879 The call:

```
8880     fid = dup2(fildes, fildes2);
```

8881 shall be equivalent to:

```
8882     close(fildes2);
```

```
8883     fid = fcntl(fildes, F_DUPFD, fildes2);
```

8884 except for the following:

- If *fildes2* is less than 0 or greater than or equal to {OPEN_MAX}, *dup2()* shall return -1 with *errno* set to [EBADF].
- If *fildes* is a valid file descriptor and is equal to *fildes2*, *dup2()* shall return *fildes2* without closing it.
- If *fildes* is not a valid file descriptor, *dup2()* shall return -1 and shall not close *fildes2*.
- The value returned shall be equal to the value of *fildes2* upon successful completion, or -1 upon failure.

8892 **RETURN VALUE**

8893 Upon successful completion a non-negative integer, namely the file descriptor, shall be returned;
8894 otherwise, -1 shall be returned and *errno* set to indicate the error.

8895 **ERRORS**

8896 The *dup()* function shall fail if:

8897 [EBADF] The *fildes* argument is not a valid open file descriptor.

8898 [EMFILE] The number of file descriptors in use by this process would exceed
8899 {OPEN_MAX}.

8900 The *dup2()* function shall fail if:

8901 [EBADF] The *fildes* argument is not a valid open file descriptor or the argument *fildes2* is
8902 negative or greater than or equal to {OPEN_MAX}.

8903 [EINTR] The *dup2()* function was interrupted by a signal.

8904 EXAMPLES

8905 Redirecting Standard Output to a File

8906 The following example closes standard output for the current processes, re-assigns standard
8907 output to go to the file referenced by *pfid*, and closes the original file descriptor to clean up.

```
8908     #include <unistd.h>
8909     ...
8910     int pfd;
8911     ...
8912     close(1);
8913     dup(pfd);
8914     close(pfd);
8915     ...
```

8916 Redirecting Error Messages

8917 The following example redirects messages from *stderr* to *stdout*.

```
8918     #include <unistd.h>
8919     ...
8920     dup2(1, 2);
8921     ...
```

8922 APPLICATION USAGE

8923 None.

8924 RATIONALE

8925 The *dup()* and *dup2()* functions are redundant. Their services are also provided by the *fcntl()*
8926 function. They have been included in this volume of IEEE Std 1003.1-2001 primarily for historical
8927 reasons, since many existing applications use them.

8928 While the brief code segment shown is very similar in behavior to *dup2()*, a conforming
8929 implementation based on other functions defined in this volume of IEEE Std 1003.1-2001 is
8930 significantly more complex. Least obvious is the possible effect of a signal-catching function that
8931 could be invoked between steps and allocate or deallocate file descriptors. This could be avoided
8932 by blocking signals.

8933 The *dup2()* function is not marked obsolescent because it presents a type-safe version of
8934 functionality provided in a type-unsafe version by *fcntl()*. It is used in the POSIX Ada binding.

8935 The *dup2()* function is not intended for use in critical regions as a synchronization mechanism.

8936 In the description of [EBADF], the case of *fildes* being out of range is covered by the given case of
8937 *fildes* not being valid. The descriptions for *fildes* and *fildes2* are different because the only kind of
8938 invalidity that is relevant for *fildes2* is whether it is out of range; that is, it does not matter
8939 whether *fildes2* refers to an open file when the *dup2()* call is made.

8940 FUTURE DIRECTIONS

8941 None.

8942 SEE ALSO

8943 *close()*, *fcntl()*, *open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

8944 **CHANGE HISTORY**

8945 First released in Issue 1. Derived from Issue 1 of the SVID.

8946 NAME

8947 ecvt, fcvt, gcvt — convert a floating-point number to a string (LEGACY)

8948 SYNOPSIS

8949 XSI #include <stdlib.h>
8950 char *ecvt(double value, int ndigit, int *restrict decpt,
8951 int *restrict sign);
8952 char *fcvt(double value, int ndigit, int *restrict decpt,
8953 int *restrict sign);
8954 char *gcvt(double value, int ndigit, char *buf);
8955

8956 DESCRIPTION

8957 The *ecvt()*, *fcvt()*, and *gcvt()* functions shall convert floating-point numbers to null-terminated
8958 strings.

8959 The *ecvt()* function shall convert *value* to a null-terminated string of *ndigit* digits (where *ndigit* is
8960 reduced to an unspecified limit determined by the precision of a **double**) and return a pointer to
8961 the string. The high-order digit shall be non-zero, unless the value is 0. The low-order digit shall
8962 be rounded in an implementation-defined manner. The position of the radix character relative to
8963 the beginning of the string shall be stored in the integer pointed to by *decpt* (negative means to
8964 the left of the returned digits). If *value* is zero, it is unspecified whether the integer pointed to by
8965 *decpt* would be 0 or 1. The radix character shall not be included in the returned string. If the sign
8966 of the result is negative, the integer pointed to by *sign* shall be non-zero; otherwise, it shall be 0.

8967 If the converted value is out of range or is not representable, the contents of the returned string
8968 are unspecified.

8969 The *fcvt()* function shall be equivalent to *ecvt()*, except that *ndigit* specifies the number of digits
8970 desired after the radix character. The total number of digits in the result string is restricted to an
8971 unspecified limit as determined by the precision of a **double**.

8972 The *gcvt()* function shall convert *value* to a null-terminated string (similar to that of the %g
8973 conversion specification format of *printf()*) in the array pointed to by *buf* and shall return *buf*. It
8974 shall produce *ndigit* significant digits (limited to an unspecified value determined by the
8975 precision of a **double**) in the %f conversion specification format of *printf()* if possible, or the %e
8976 conversion specification format of *printf()* (scientific notation) otherwise. A minus sign shall be
8977 included in the returned string if *value* is less than 0. A radix character shall be included in the
8978 returned string if *value* is not a whole number. Trailing zeros shall be suppressed where *value* is
8979 not a whole number. The radix character is determined by the current locale. If *setlocale()* has not
8980 been called successfully, the default locale, POSIX, is used. The default locale specifies a period
8981 ('.') as the radix character. The *LC_NUMERIC* category determines the value of the radix
8982 character within the current locale.

8983 These functions need not be reentrant. A function that is not required to be reentrant is not
8984 required to be thread-safe.

8985 RETURN VALUE

8986 The *ecvt()* and *fcvt()* functions shall return a pointer to a null-terminated string of digits.

8987 The *gcvt()* function shall return *buf*.

8988 The return values from *ecvt()* and *fcvt()* may point to static data which may be overwritten by
8989 subsequent calls to these functions.

8990 ERRORS

8991 No errors are defined.

8992 EXAMPLES

8993 None.

8994 APPLICATION USAGE

8995 The *sprintf()* function is preferred over this function.

8996 RATIONALE

8997 None.

8998 FUTURE DIRECTIONS

8999 These functions may be withdrawn in a future version.

9000 SEE ALSO

9001 *printf()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

9002 CHANGE HISTORY

9003 First released in Issue 4, Version 2.

9004 Issue 5

9005 Moved from X/OPEN UNIX extension to BASE.

9006 Normative text previously in the APPLICATION USAGE section is moved to the
9007 DESCRIPTION.

9008 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

9009 Issue 6

9010 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

9011 This function is marked LEGACY.

9012 The **restrict** keyword is added to the *ecvt()* and *fcvt()* prototypes for alignment with the
9013 ISO/IEC 9899: 1999 standard.

9014 The DESCRIPTION is updated to explicitly use “conversion specification” to describe %g, %f,
9015 and %e.

9016 NAME

9017 encrypt — encoding function (CRYPT)

9018 SYNOPSIS

9019 XSI #include <unistd.h>
9020 void encrypt(char block[64], int edflag);
9021

9022 DESCRIPTION

9023 The *encrypt()* function shall provide access to an implementation-defined encoding algorithm.
9024 The key generated by *setkey()* is used to encrypt the string *block* with *encrypt()*.

9025 The *block* argument to *encrypt()* shall be an array of length 64 bytes containing only the bytes
9026 with values of 0 and 1. The array is modified in place to a similar array using the key set by
9027 *setkey()*. If *edflag* is 0, the argument is encoded. If *edflag* is 1, the argument may be decoded (see
9028 the APPLICATION USAGE section); if the argument is not decoded, *errno* shall be set to
9029 [ENOSYS].

9030 The *encrypt()* function shall not change the setting of *errno* if successful. An application wishing
9031 to check for error situations should set *errno* to 0 before calling *encrypt()*. If *errno* is non-zero on
9032 return, an error has occurred.

9033 The *encrypt()* function need not be reentrant. A function that is not required to be reentrant is
9034 not required to be thread-safe.

9035 RETURN VALUE

9036 The *encrypt()* function shall not return a value.

9037 ERRORS

9038 The *encrypt()* function shall fail if:

9039 [ENOSYS] The functionality is not supported on this implementation.

9040 EXAMPLES

9041 None.

9042 APPLICATION USAGE

9043 Historical implementations of the *encrypt()* function used a rather primitive encoding algorithm.

9044 In some environments, decoding might not be implemented. This is related to some Government
9045 restrictions on encryption and decryption routines. Historical practice has been to ship a
9046 different version of the encryption library without the decryption feature in the routines
9047 supplied. Thus the exported version of *encrypt()* does encoding but not decoding.

9048 RATIONALE

9049 None.

9050 FUTURE DIRECTIONS

9051 None.

9052 SEE ALSO

9053 *crypt()*, *setkey()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

9054 CHANGE HISTORY

9055 First released in Issue 1. Derived from Issue 1 of the SVID.

9056 **Issue 5**

9057 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

9058 **Issue 6**

9059 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

9060 NAME

9061 endgrent, getgrent, setgrent — group database entry functions

9062 SYNOPSIS

9063 XSI #include <grp.h>

```
9064            void endgrent(void);
9065            struct group *getgrent(void);
9066            void setgrent(void);
```

9068 DESCRIPTION

9069 The *getgrent()* function shall return a pointer to a structure containing the broken-out fields of an
9070 entry in the group database. When first called, *getgrent()* shall return a pointer to a **group**
9071 structure containing the first entry in the group database. Thereafter, it shall return a pointer to a
9072 **group** structure containing the next group structure in the group database, so successive calls
9073 may be used to search the entire database.

9074 An implementation that provides extended security controls may impose further
9075 implementation-defined restrictions on accessing the group database. In particular, the system
9076 may deny the existence of some or all of the group database entries associated with groups other
9077 than those groups associated with the caller and may omit users other than the caller from the
9078 list of members of groups in database entries that are returned.

9079 The *setgrent()* function shall rewind the group database to allow repeated searches.

9080 The *endgrent()* function may be called to close the group database when processing is complete.

9081 These functions need not be reentrant. A function that is not required to be reentrant is not
9082 required to be thread-safe.

9083 RETURN VALUE

9084 When first called, *getgrent()* shall return a pointer to the first group structure in the group
9085 database. Upon subsequent calls it shall return the next group structure in the group database.
9086 The *getgrent()* function shall return a null pointer on end-of-file or an error and *errno* may be set
9087 to indicate the error.

9088 The return value may point to a static area which is overwritten by a subsequent call to
9089 *getgrgid()*, *getgrnam()*, or *getgrent()*.

9090 ERRORS

9091 The *getgrent()* function may fail if:

9092 [EINTR]	A signal was caught during the operation.
9093 [EIO]	An I/O error has occurred.
9094 [EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
9095 [ENFILE]	The maximum allowable number of files is currently open in the system.

9096 EXAMPLES

9097 None.

9098 APPLICATION USAGE

9099 These functions are provided due to their historical usage. Applications should avoid
9100 dependencies on fields in the group database, whether the database is a single file, or where in
9101 the file system name space the database resides. Applications should use *getgrnam()* and
9102 *getgrgid()* whenever possible because it avoids these dependencies.

9103 RATIONALE

9104 None.

9105 FUTURE DIRECTIONS

9106 None.

9107 SEE ALSO

9108 *getgrgid()*, *getgrnam()*, *getlogin()*, *getpwent()*, the Base Definitions volume of
9109 IEEE Std 1003.1-2001, <grp.h>

9110 CHANGE HISTORY

9111 First released in Issue 4, Version 2.

9112 Issue 5

9113 Moved from X/OPEN UNIX extension to BASE.

9114 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
9115 VALUE section.

9116 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

9117 Issue 6

9118 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

9119 NAME

9120 endhostent, gethostent, sethostent — network host database functions

9121 SYNOPSIS

```
9122     #include <netdb.h>
9123
9124     void endhostent(void);
9125     struct hostent *gethostent(void);
9126     void sethostent(int stayopen);
```

9126 DESCRIPTION

9127 These functions shall retrieve information about hosts. This information is considered to be
9128 stored in a database that can be accessed sequentially or randomly. The implementation of this
9129 database is unspecified.

9130 **Note:** In many cases this database is implemented by the Domain Name System, as documented in
9131 RFC 1034, RFC 1035, and RFC 1886.

9132 The *sethostent()* function shall open a connection to the database and set the next entry for
9133 retrieval to the first entry in the database. If the *stayopen* argument is non-zero, the connection
9134 shall not be closed by a call to *gethostent()*, *gethostbyname()*, or *gethostbyaddr()*, and the
9135 implementation may maintain an open file descriptor.

9136 The *gethostent()* function shall read the next entry in the database, opening and closing a
9137 connection to the database as necessary.

9138 Entries shall be returned in **hostent** structures. Refer to *gethostbyaddr()* for a definition of the
9139 **hostent** structure.

9140 The *endhostent()* function shall close the connection to the database, releasing any open file
9141 descriptor.

9142 These functions need not be reentrant. A function that is not required to be reentrant is not
9143 required to be thread-safe.

9144 RETURN VALUE

9145 Upon successful completion, the *gethostent()* function shall return a pointer to a **hostent**
9146 structure if the requested entry was found, and a null pointer if the end of the database was
9147 reached or the requested entry was not found.

9148 ERRORS

9149 No errors are defined for *endhostent()*, *gethostent()*, and *sethostent()*.

9150 EXAMPLES

9151 None.

9152 APPLICATION USAGE

9153 The *gethostent()* function may return pointers to static data, which may be overwritten by
9154 subsequent calls to any of these functions.

9155 RATIONALE

9156 None.

9157 FUTURE DIRECTIONS

9158 None.

9159 SEE ALSO

9160 *endservent()*, *gethostbyaddr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>

9161 CHANGE HISTORY

9162 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

9163 **NAME**

9164 endnetent, getnetbyaddr, getnetbyname, getnetent, setnetent — network database functions

9165 **SYNOPSIS**

```
9166     #include <netdb.h>
9167
9168     void endnetent(void);
9169     struct netent *getnetbyaddr(uint32_t net, int type);
9170     struct netent *getnetbyname(const char *name);
9171     struct netent *getnetent(void);
9172     void setnetent(int stayopen);
```

9172 **DESCRIPTION**

9173 These functions shall retrieve information about networks. This information is considered to be
9174 stored in a database that can be accessed sequentially or randomly. The implementation of this
9175 database is unspecified.

9176 The *setnetent()* function shall open and rewind the database. If the *stayopen* argument is non-
9177 zero, the connection to the *net* database shall not be closed after each call to *getnetent()* (either
9178 directly, or indirectly through one of the other *getnet**() functions), and the implementation may
9179 maintain an open file descriptor to the database.

9180 The *getnetent()* function shall read the next entry of the database, opening and closing a
9181 connection to the database as necessary.

9182 The *getnetbyaddr()* function shall search the database from the beginning, and find the first entry
9183 for which the address family specified by *type* matches the *n_addrtype* member and the network
9184 number *net* matches the *n_net* member, opening and closing a connection to the database as
9185 necessary. The *net* argument shall be the network number in host byte order.

9186 The *getnetbyname()* function shall search the database from the beginning and find the first entry
9187 for which the network name specified by *name* matches the *n_name* member, opening and
9188 closing a connection to the database as necessary.

9189 The *getnetbyaddr()*, *getnetbyname()*, and *getnetent()* functions shall each return a pointer to a
9190 **netent** structure, the members of which shall contain the fields of an entry in the network
9191 database.

9192 The *endnetent()* function shall close the database, releasing any open file descriptor.

9193 These functions need not be reentrant. A function that is not required to be reentrant is not
9194 required to be thread-safe.

9195 **RETURN VALUE**

9196 Upon successful completion, *getnetbyaddr()*, *getnetbyname()*, and *getnetent()* shall return a
9197 pointer to a **netent** structure if the requested entry was found, and a null pointer if the end of the
9198 database was reached or the requested entry was not found. Otherwise, a null pointer shall be
9199 returned.

9200 **ERRORS**

9201 No errors are defined.

9202 **EXAMPLES**

9203 None.

9204 **APPLICATION USAGE**9205 The *getnetbyaddr()*, *getnetbyname()*, and *getnetent()* functions may return pointers to static data,
9206 which may be overwritten by subsequent calls to any of these functions.9207 **RATIONALE**

9208 None.

9209 **FUTURE DIRECTIONS**

9210 None.

9211 **SEE ALSO**9212 The Base Definitions volume of IEEE Std 1003.1-2001, <**netdb.h**>9213 **CHANGE HISTORY**

9214 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

9215 **NAME**

9216 endprotoent, getprotobynumber, getprotoent, setprotoent — network protocol
9217 database functions

9218 **SYNOPSIS**

```
9219     #include <netdb.h>
9220
9221     void endprotoent(void);
9222     struct protoent *getprotobynumber(const char *name);
9223     struct protoent *getprotoent(int proto);
9224     struct protoent *getprotobynumber(int proto);
9225     void setprotoent(int stayopen);
```

9225 **DESCRIPTION**

9226 These functions shall retrieve information about protocols. This information is considered to be
9227 stored in a database that can be accessed sequentially or randomly. The implementation of this
9228 database is unspecified.

9229 The *setprotoent()* function shall open a connection to the database, and set the next entry to the
9230 first entry. If the *stayopen* argument is non-zero, the connection to the network protocol database
9231 shall not be closed after each call to *getprotoent()* (either directly, or indirectly through one of the
9232 other *getproto**() functions), and the implementation may maintain an open file descriptor for
9233 the database.

9234 The *getprotobynumber()* function shall search the database from the beginning and find the first
9235 entry for which the protocol number specified by *proto* matches the *p_proto* member, opening and
9236 closing a connection to the database as necessary.

9237 The *getprotoent()* function shall search the database from the beginning and find the first
9238 entry for which the protocol name specified by *name* matches the *p_name* member, opening and
9239 closing a connection to the database as necessary.

9240 The *getprotoent()* function shall read the next entry of the database, opening and closing a
9241 connection to the database as necessary.

9242 The *getprotoent()*, *getprotobynumber()*, and *getprotoent()* functions shall each return a pointer
9243 to a **protoent** structure, the members of which shall contain the fields of an entry in the network
9244 protocol database.

9245 The *endprotoent()* function shall close the connection to the database, releasing any open file
9246 descriptor.

9247 These functions need not be reentrant. A function that is not required to be reentrant is not
9248 required to be thread-safe.

9249 **RETURN VALUE**

9250 Upon successful completion, *getprotobynumber()*, *getprotobynumber()*, and *getprotoent()* return a
9251 pointer to a **protoent** structure if the requested entry was found, and a null pointer if the end of
9252 the database was reached or the requested entry was not found. Otherwise, a null pointer is
9253 returned.

9254 **ERRORS**

9255 No errors are defined.

9256 EXAMPLES

9257 None.

9258 APPLICATION USAGE

9259 The *getprotobynumber()*, *getprotobynumber()*, and *getprotoent()* functions may return pointers to
9260 static data, which may be overwritten by subsequent calls to any of these functions.

9261 RATIONALE

9262 None.

9263 FUTURE DIRECTIONS

9264 None.

9265 SEE ALSO

9266 The Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>

9267 CHANGE HISTORY

9268 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

9269 **NAME**

9270 endpwent, getpwent, setpwent — user database functions

9271 **SYNOPSIS**

9272 XSI #include <pwd.h>

```
9273           void endpwent(void);
9274           struct passwd *getpwent(void);
9275           void setpwent(void);
```

9277 **DESCRIPTION**

9278 These functions shall retrieve information about users.

9279 The *getpwent()* function shall return a pointer to a structure containing the broken-out fields of
9280 an entry in the user database. Each entry in the user database contains a **passwd** structure. When
9281 first called, *getpwent()* shall return a pointer to a **passwd** structure containing the first entry in
9282 the user database. Thereafter, it shall return a pointer to a **passwd** structure containing the next
9283 entry in the user database. Successive calls can be used to search the entire user database.

9284 If an end-of-file or an error is encountered on reading, *getpwent()* shall return a null pointer.

9285 An implementation that provides extended security controls may impose further
9286 implementation-defined restrictions on accessing the user database. In particular, the system
9287 may deny the existence of some or all of the user database entries associated with users other
9288 than the caller.

9289 The *setpwent()* function effectively rewinds the user database to allow repeated searches.

9290 The *endpwent()* function may be called to close the user database when processing is complete.

9291 These functions need not be reentrant. A function that is not required to be reentrant is not
9292 required to be thread-safe.

9293 **RETURN VALUE**

9294 The *getpwent()* function shall return a null pointer on end-of-file or error.

9295 **ERRORS**

9296 The *getpwent()*, *setpwent()*, and *endpwent()* functions may fail if:

9297 [EIO] An I/O error has occurred.

9298 In addition, *getpwent()* and *setpwent()* may fail if:

9299 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

9300 [ENFILE] The maximum allowable number of files is currently open in the system.

9301 The return value may point to a static area which is overwritten by a subsequent call to
9302 *getpwuid()*, *getpwnam()*, or *getpwent()*.

9303 EXAMPLES

9304 **Searching the User Database**

9305 The following example uses the *getpwent()* function to get successive entries in the user
9306 database, returning a pointer to a **passwd** structure that contains information about each user.
9307 The call to *endpwent()* closes the user database and cleans up.

```
9308 #include <pwd.h>
9309 ...
9310 struct passwd *p;
9311 ...
9312 while ((p = getpwent ()) != NULL) {
9313 ...
9314 }
9315 endpwent ();
9316 ...
```

9317 **APPLICATION USAGE**

9318 These functions are provided due to their historical usage. Applications should avoid
9319 dependencies on fields in the password database, whether the database is a single file, or where
9320 in the file system name space the database resides. Applications should use *getpwuid()*
9321 whenever possible because it avoids these dependencies.

9322 **RATIONALE**

9323 None.

9324 **FUTURE DIRECTIONS**

9325 None.

9326 **SEE ALSO**

9327 *endrent()*, *getlogin()*, *getpwnam()*, *getpwuid()*, the Base Definitions volume of
9328 IEEE Std 1003.1-2001, <**pwd.h**>

9329 **CHANGE HISTORY**

9330 First released in Issue 4, Version 2.

9331 **Issue 5**

9332 Moved from X/OPEN UNIX extension to BASE.

9333 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
9334 VALUE section.

9335 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

9336 **Issue 6**

9337 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

9338 **NAME**

9339 endservent, getservbyname, getservbyport, getservent, setservent — network services database
9340 functions

9341 **SYNOPSIS**

```
9342     #include <netdb.h>
9343
9344     void endservent(void);
9345     struct servent *getservbyname(const char *name, const char *proto);
9346     struct servent *getservbyport(int port, const char *proto);
9347     struct servent *getservent(void);
9348     void setservent(int stayopen);
```

9348 **DESCRIPTION**

9349 These functions shall retrieve information about network services. This information is
9350 considered to be stored in a database that can be accessed sequentially or randomly. The
9351 implementation of this database is unspecified.

9352 The *setservent()* function shall open a connection to the database, and set the next entry to the
9353 first entry. If the *stayopen* argument is non-zero, the *net* database shall not be closed after each
9354 call to the *getservent()* function (either directly, or indirectly through one of the other *getserv**()
9355 functions), and the implementation may maintain an open file descriptor for the database.

9356 The *getservent()* function shall read the next entry of the database, opening and closing a
9357 connection to the database as necessary.

9358 The *getservbyname()* function shall search the database from the beginning and find the first
9359 entry for which the service name specified by *name* matches the *s_name* member and the protocol
9360 name specified by *proto* matches the *s_proto* member, opening and closing a connection to the
9361 database as necessary. If *proto* is a null pointer, any value of the *s_proto* member shall be
9362 matched.

9363 The *getservbyport()* function shall search the database from the beginning and find the first entry
9364 for which the port specified by *port* matches the *s_port* member and the protocol name specified
9365 by *proto* matches the *s_proto* member, opening and closing a connection to the database as
9366 necessary. If *proto* is a null pointer, any value of the *s_proto* member shall be matched. The *port*
9367 argument shall be in network byte order.

9368 The *getservbyname()*, *getservbyport()*, and *getservent()* functions shall each return a pointer to a
9369 **servent** structure, the members of which shall contain the fields of an entry in the network
9370 services database.

9371 The *endservent()* function shall close the database, releasing any open file descriptor.

9372 These functions need not be reentrant. A function that is not required to be reentrant is not
9373 required to be thread-safe.

9374 **RETURN VALUE**

9375 Upon successful completion, *getservbyname()*, *getservbyport()*, and *getservent()* return a pointer to
9376 a **servent** structure if the requested entry was found, and a null pointer if the end of the database
9377 was reached or the requested entry was not found. Otherwise, a null pointer is returned.

9378 **ERRORS**

9379 No errors are defined.

9380 EXAMPLES

9381 None.

9382 APPLICATION USAGE

9383 The *port* argument of *getservbyport()* need not be compatible with the port values of all address families.

9385 The *getservbyname()*, *getservbyport()*, and *getservent()* functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

9387 RATIONALE

9388 None.

9389 FUTURE DIRECTIONS

9390 None.

9391 SEE ALSO

9392 *endhostent()*, *endprotoent()*, *htonl()*, *inet_addr()*, the Base Definitions volume of
9393 IEEE Std 1003.1-2001, <netdb.h>

9394 CHANGE HISTORY

9395 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

9396 NAME

9397 endutxent, getutxent, getutxid, getutxline, pututxline, setutxent — user accounting database
9398 functions

9399 SYNOPSIS

9400 XSI

```
#include <utmpx.h>

void endutxent(void);
struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
struct utmpx *pututxline(const struct utmpx *utmpx);
void setutxent(void);
```

9407

9408 DESCRIPTION

9409 These functions shall provide access to the user accounting database.

9410 The *getutxent()* function shall read the next entry from the user accounting database. If the
9411 database is not already open, it shall open it. If it reaches the end of the database, it shall fail.

9412 The *getutxid()* function shall search forward from the current point in the database. If the
9413 *ut_type* value of the **utmpx** structure pointed to by *id* is *BOOT_TIME*, *OLD_TIME*, or
9414 *NEW_TIME*, then it shall stop when it finds an entry with a matching *ut_type* value. If the
9415 *ut_type* value is *INIT_PROCESS*, *LOGIN_PROCESS*, *USER_PROCESS*, or *DEAD_PROCESS*,
9416 then it shall stop when it finds an entry whose type is one of these four and whose *ut_id* member
9417 matches the *ut_id* member of the **utmpx** structure pointed to by *id*. If the end of the database is
9418 reached without a match, *getutxid()* shall fail.

9419 The *getutxline()* function shall search forward from the current point in the database until it
9420 finds an entry of the type *LOGIN_PROCESS* or *USER_PROCESS* which also has a *ut_line* value
9421 matching that in the **utmpx** structure pointed to by *line*. If the end of the database is reached
9422 without a match, *getutxline()* shall fail.

9423 The *getutxid()* or *getutxline()* function may cache data. For this reason, to use *getutxline()* to
9424 search for multiple occurrences, the application shall zero out the static data after each success,
9425 or *getutxline()* may return a pointer to the same **utmpx** structure.

9426 There is one exception to the rule about clearing the structure before further reads are done. The
9427 implicit read done by *pututxline()* (if it finds that it is not already at the correct place in the user
9428 accounting database) shall not modify the static structure returned by *getutxent()*, *getutxid()*, or
9429 *getutxline()*, if the application has modified this structure and passed the pointer back to
9430 *pututxline()*.

9431 For all entries that match a request, the *ut_type* member indicates the type of the entry. Other
9432 members of the entry shall contain meaningful data based on the value of the *ut_type* member as
9433 follows:

9434
9435
9436
9437
9438
9439
9440
9441
9442
9443
9444

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>

9445 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.

9446 If the process has appropriate privileges, the *pututxline()* function shall write out the structure
9447 into the user accounting database. It shall use *getutxid()* to search for a record that satisfies the
9448 request. If this search succeeds, then the entry shall be replaced. Otherwise, a new entry shall be
9449 made at the end of the user accounting database.

9450 The *endutxent()* function shall close the user accounting database.
9451

9452 The *setutxent()* function shall reset the input to the beginning of the database. This should be
9453 done before each search for a new entry if it is desired that the entire database be examined.
9454

9455 These functions need not be reentrant. A function that is not required to be reentrant is not
9456 required to be thread-safe.
9457

9458 RETURN VALUE

9459 Upon successful completion, *getutxent()*, *getutxid()*, and *getutxline()* shall return a pointer to a
9460 **utmpx** structure containing a copy of the requested entry in the user accounting database.
9461 Otherwise, a null pointer shall be returned.

9462 The return value may point to a static area which is overwritten by a subsequent call to
9463 *getutxid()* or *getutxline()*.
9464

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.
9466

The *endutxent()* and *setutxent()* functions shall not return a value.
9467

9468 ERRORS

9469 No errors are defined for the *endutxent()*, *getutxent()*, *getutxid()*, *getutxline()*, and *setutxent()*
9470 functions.

9471 The *pututxline()* function may fail if:
9472

[EPERM] The process does not have appropriate privileges.

9473 EXAMPLES

9474 None.

9475 APPLICATION USAGE

9476 The sizes of the arrays in the structure can be found using the *sizeof* operator.

9477 RATIONALE

9478 None.

9479 FUTURE DIRECTIONS

9480 None.

9481 SEE ALSO

9482 The Base Definitions volume of IEEE Std 1003.1-2001, <utmpx.h>

9483 CHANGE HISTORY

9484 First released in Issue 4, Version 2.

9485 Issue 5

9486 Moved from X/OPEN UNIX extension to BASE.

9487 Normative text previously in the APPLICATION USAGE section is moved to the
9488 DESCRIPTION.

9489 A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

9490 Issue 6

9491 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

9492 **NAME**9493 **environ** — array of character pointers to the environment strings9494 **SYNOPSIS**9495 **extern char **environ;**9496 **DESCRIPTION**9497 Refer to the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables
9498 and **exec**.

9499 NAME

9500 erand48 — generate uniformly distributed pseudo-random numbers

9501 SYNOPSIS

9502 XSI #include <stdlib.h>

9503 double erand48(unsigned short xsubi[3]);

9504

9505 DESCRIPTION

9506 Refer to *drand48()*.

9507 **NAME**

9508 erf, erff, erfl — error functions

9509 **SYNOPSIS**

```
9510     #include <math.h>
9511
9512     double erf(double x);
9513     float erff(float x);
9514     long double erfl(long double x);
```

9514 **DESCRIPTION**

9515 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 9516 conflict between the requirements described here and the ISO C standard is unintentional. This
 9517 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

9518 These functions shall compute the error function of their argument *x*, defined as:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

9520 An application wishing to check for error situations should set *errno* to zero and call
 9521 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
 9522 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
 9523 zero, an error has occurred.

9524 **RETURN VALUE**

9525 Upon successful completion, these functions shall return the value of the error function.

9526 MX If *x* is NaN, a NaN shall be returned.

9527 If *x* is ±0, ±0 shall be returned.

9528 If *x* is ±Inf, ±1 shall be returned.

9529 If *x* is subnormal, a range error may occur, and $2 * x / \sqrt{\pi}$ should be returned.

9530 **ERRORS**

9531 These functions may fail if:

9532 MX Range Error The result underflows.

9533 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 9534 then *errno* shall be set to [ERANGE]. If the integer expression
 9535 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 9536 floating-point exception shall be raised.

9537 **EXAMPLES**9538 **Computing the Probability for a Normal Variate**

2

9539 This example shows how to use *erf()* to compute the probability that a normal variate assumes a
 9540 value in the range [*x1*,*x2*] with *x1*≤*x2*. This example uses the constant M_SQRT1_2 which is an
 9541 XSI extension.

```
9542 #include <math.h>
9543
9544 double
9545 Phi(const double x1, const double x2)
9546 {
9547     return ( erf(x2*M_SQRT1_2) - erf(x1*M_SQRT1_2) ) / 2;
9548 }
```

2 2 2 2 2 2 2 2

9548 **APPLICATION USAGE**

9549 Underflow occurs when $|x| < \text{DBL_MIN} * (\sqrt{\pi})/2$.

9550 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
9551 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

9552 **RATIONALE**

9553 None.

9554 **FUTURE DIRECTIONS**

9555 None.

9556 **SEE ALSO**

9557 *erfc()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
9558 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

9559 **CHANGE HISTORY**

9560 First released in Issue 1. Derived from Issue 1 of the SVID.

9561 **Issue 5**

9562 The DESCRIPTION is updated to indicate how an application should check for an error. This
9563 text was previously published in the APPLICATION USAGE section.

9564 **Issue 6**

9565 The *erf()* function is no longer marked as an extension.

9566 The *erfc()* function is split out onto its own reference page.

9567 The *erff()* and *erfl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

9568 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
9569 revised to align with the ISO/IEC 9899:1999 standard.

9570 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
9571 marked.

9572 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/22 is applied, adding the example to the
9573 EXAMPLES section. 2 2

9574 **NAME**

9575 erfc, erfcf, erfcl — complementary error functions

9576 **SYNOPSIS**

```
9577     #include <math.h>
9578
9579     double erfc(double x);
9580     float erfcf(float x);
9581     long double erfcl(long double x);
```

9581 **DESCRIPTION**

9582 CX The functionality described on this reference page is aligned with the ISO C standard. Any
9583 conflict between the requirements described here and the ISO C standard is unintentional. This
9584 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

9585 These functions shall compute the complementary error function $1.0 - \text{erf}(x)$.

9586 An application wishing to check for error situations should set *errno* to zero and call
9587 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
9588 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
9589 zero, an error has occurred.

9590 **RETURN VALUE**

9591 Upon successful completion, these functions shall return the value of the complementary error
9592 function.

9593 If the correct value would cause underflow and is not representable, a range error may occur
9594 MX and either 0.0 (if representable), or an implementation-defined value shall be returned.

9595 MX If *x* is NaN, a NaN shall be returned.

9596 If *x* is ± 0 , +1 shall be returned.

9597 If *x* is $-\text{Inf}$, +2 shall be returned.

9598 If *x* is $+\text{Inf}$, +0 shall be returned.

9599 If the correct value would cause underflow and is representable, a range error may occur and the
9600 correct value shall be returned.

9601 **ERRORS**

9602 These functions may fail if:

9603 Range Error The result underflows.

9604 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
9605 then *errno* shall be set to [ERANGE]. If the integer expression
9606 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
9607 floating-point exception shall be raised.

9608 **EXAMPLES**

9609 None.

9610 **APPLICATION USAGE**

9611 The *erfc()* function is provided because of the extreme loss of relative accuracy if *erf(x)* is called
9612 for large *x* and the result subtracted from 1.0.

9613 Note for IEEE Std 754-1985 **double**, $26.55 < x$ implies *erfc(x)* has underflowed.

9614 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
9615 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

9616 RATIONALE

9617 None.

9618 FUTURE DIRECTIONS

9619 None.

9620 SEE ALSO

9621 *erf()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
9622 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

9623 CHANGE HISTORY

9624 First released in Issue 1. Derived from Issue 1 of the SVID.

9625 Issue 5

9626 The DESCRIPTION is updated to indicate how an application should check for an error. This
9627 text was previously published in the APPLICATION USAGE section.

9628 Issue 6

9629 The *erfc()* function is no longer marked as an extension.

9630 These functions are split out from the *erf()* reference page.

9631 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
9632 revised to align with the ISO/IEC 9899:1999 standard.

9633 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
9634 marked.

9635 **NAME**
9636 erff, erfl — error functions

9637 **SYNOPSIS**
9638 #include <math.h>
9639 float erff(float x);
9640 long double erfl(long double x);

9641 **DESCRIPTION**
9642 Refer to *erf()*.

9643 NAME

9644 `errno` — error return value

9645 SYNOPSIS

9646 `#include <errno.h>`

9647 DESCRIPTION

9648 The lvalue `errno` is used by many functions to return error values.

9649 Many functions provide an error number in `errno`, which has type `int` and is defined in
9650 `<errno.h>`. The value of `errno` shall be defined only after a call to a function for which it is
9651 explicitly stated to be set and until it is changed by the next function call or if the application
9652 assigns it a value. The value of `errno` should only be examined when it is indicated to be valid by
9653 a function's return value. Applications shall obtain the definition of `errno` by the inclusion of
9654 `<errno.h>`. No function in this volume of IEEE Std 1003.1-2001 shall set `errno` to 0. The setting of
9655 `errno` after a successful call to a function is unspecified unless the description of that function
9656 specifies that `errno` shall not be modified.

2
2
2

9657 It is unspecified whether `errno` is a macro or an identifier declared with external linkage. If a
9658 macro definition is suppressed in order to access an actual object, or a program defines an
9659 identifier with the name `errno`, the behavior is undefined.

9660 The symbolic values stored in `errno` are documented in the ERRORS sections on all relevant
9661 pages.

9662 RETURN VALUE

9663 None.

9664 ERRORS

9665 None.

9666 EXAMPLES

9667 None.

9668 APPLICATION USAGE

9669 Previously both POSIX and X/Open documents were more restrictive than the ISO C standard
9670 in that they required `errno` to be defined as an external variable, whereas the ISO C standard
9671 required only that `errno` be defined as a modifiable lvalue with type `int`.

9672 An application that needs to examine the value of `errno` to determine the error should set it to 0
9673 before a function call, then inspect it before a subsequent function call.

9674 RATIONALE

9675 None.

9676 FUTURE DIRECTIONS

9677 None.

9678 SEE ALSO

9679 Section 2.3, the Base Definitions volume of IEEE Std 1003.1-2001, `<errno.h>`

9680 CHANGE HISTORY

9681 First released in Issue 1. Derived from Issue 1 of the SVID.

9682 Issue 5

9683 The following sentence is deleted from the DESCRIPTION: "The value of `errno` is 0 at program
9684 start-up, but is never set to 0 by any XSI function". The DESCRIPTION also no longer states that
9685 conforming implementations may support the declaration:

9686 extern int errno;

9687 **Issue 6**

9688 Obsolescent text regarding defining *errno* as:

9689 extern int errno

9690 is removed.

9691 Text regarding no function setting *errno* to zero to indicate an error is changed to no function
9692 shall set *errno* to zero. This is for alignment with the ISO/IEC 9899:1999 standard.

9693 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/23 is applied, adding text to the 2
9694 DESCRIPTION stating that the setting of *errno* after a successful call to a function is unspecified 2
9695 unless the description of the function requires that it will not be modified. 2

9696 NAME

9697 environ, execl, execv, execle, execve, execlp, execvp — execute a file

9698 SYNOPSIS

```
9699     #include <unistd.h>
9700
9701     extern char **environ;
9702     int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
9703     int execv(const char *path, char *const argv[]);
9704     int execle(const char *path, const char *arg0, ... /*,
9705                 (char *)0, char *const envp[]*/);
9706     int execve(const char *path, char *const argv[], char *const envp[]);
9707     int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
9708     int execvp(const char *file, char *const argv[]);
```

9708 DESCRIPTION

9709 The *exec* family of functions shall replace the current process image with a new process image.
 9710 The new image shall be constructed from a regular, executable file called the *new process image file*. There shall be no return from a successful *exec*, because the calling process image is overlaid by the new process image.

9713 When a C-language program is executed as a result of this call, it shall be entered as a C-language function call as follows:

```
9715     int main (int argc, char *argv[]);
```

9716 where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
9718     extern char **environ;
```

9719 is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv* array is not counted in *argc*.

9722 THR Conforming multi-threaded applications shall not use the *environ* variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable shall be considered a use of the *environ* variable to access that environment variable.

9726 The arguments specified by a program with one of the *exec* functions shall be passed on to the new process image in the corresponding *main()* arguments.

9728 The argument *path* points to a pathname that identifies the new process image file.

9729 The argument *file* is used to construct a pathname that identifies the new process image file. If the *file* argument contains a slash character, the *file* argument shall be used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable *PATH* (see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables). If this environment variable is not present, the results of the search are implementation-defined.

9735 There are two distinct ways in which the contents of the process image file may cause the execution to fail, distinguished by the setting of *errno* to either [ENOEXEC] or [EINVAL] (see the ERRORS section). In the cases where the other members of the *exec* family of functions would fail and set *errno* to [ENOEXEC], the *execl()* and *execvp()* functions shall execute a command interpreter and the environment of the executed command shall be as if the process invoked the *sh* utility using *exec()* as follows:

```

9741     execl(<shell path>, arg0, file, arg1, ..., (char *)0);
9742 where <shell path> is an unspecified pathname for the sh utility, file is the process image file, and
9743 for execvp(), where arg0, arg1, and so on correspond to the values passed to execvp() in argv[0],
9744 argv[1], and so on.

9745 The arguments represented by arg0,... are pointers to null-terminated character strings. These
9746 strings shall constitute the argument list available to the new process image. The list is
9747 terminated by a null pointer. The argument arg0 should point to a filename that is associated
9748 with the process being started by one of the exec functions.

9749 The argument argv is an array of character pointers to null-terminated strings. The application
9750 shall ensure that the last member of this array is a null pointer. These strings shall constitute the
9751 argument list available to the new process image. The value in argv[0] should point to a filename
9752 that is associated with the process being started by one of the exec functions.

9753 The argument envp is an array of character pointers to null-terminated strings. These strings
9754 shall constitute the environment for the new process image. The envp array is terminated by a
9755 null pointer.

9756 For those forms not containing an envp pointer (execl(), execv(), execlp(), and execvp()), the
9757 environment for the new process image shall be taken from the external variable environ in the
9758 calling process.

9759 The number of bytes available for the new process' combined argument and environment lists is
9760 {ARG_MAX}. It is implementation-defined whether null terminators, pointers, and/or any
9761 alignment bytes are included in this total.

9762 File descriptors open in the calling process image shall remain open in the new process image,
9763 except for those whose close-on-exec flag FD_CLOEXEC is set. For those file descriptors that
9764 remain open, all attributes of the open file description remain unchanged. For any file descriptor
9765 that is closed for this reason, file locks are removed as a result of the close as described in close().
9766 Locks that are not removed by closing of file descriptors remain unchanged.

9767 If file descriptors 0, 1, and 2 would otherwise be closed after a successful call to one of the exec 1
9768 family of functions, and the new process image file has the set-user-ID or set-group-ID file mode 1
9769 XSI bits set, and the ST_NOSUID bit is not set for the file system containing the new process image 1
9770 file, implementations may open an unspecified file for each of these file descriptors in the new 1
9771 process image. 1

9772 Directory streams open in the calling process image shall be closed in the new process image.

9773 THR The state of the floating-point environment in the new process image or in the initial thread of 2
9774 the new process image shall be set to the default. 2

9775 XSI The state of conversion descriptors and message catalog descriptors in the new process image is 2
9776 undefined. For the new process image, the equivalent of:
9777
9778     setlocale(LC_ALL, "C")
9779
9780 Signals set to the default action (SIG_DFL) in the calling process image shall be set to the default 2
9781 action in the new process image. Except for SIGCHLD, signals set to be ignored (SIG_IGN) by 2
9782 the calling process image shall be set to be ignored by the new process image. Signals set to be 2
9783 caught by the calling process image shall be set to the default action in the new process image 2
9784 (see <signal.h>). If the SIGCHLD signal is set to be ignored by the calling process image, it is 2
9785 XSI unspecified whether the SIGCHLD signal is set to be ignored or to the default action in the new 2

```

9786	process image. After a successful call to any of the <code>exec</code> functions, alternate signal stacks are not preserved and the <code>SA_ONSTACK</code> flag shall be cleared for all signals.	
9788	THR After a successful call to any of the <code>exec</code> functions, any functions previously registered by <code>atexit()</code> or <code>pthread_atfork()</code> are no longer registered.	2
9789		2
9790	XSI If the <code>ST_NOSUID</code> bit is set for the file system containing the new process image file, then the effective user ID, effective group ID, saved set-user-ID, and saved set-group-ID are unchanged in the new process image. Otherwise, if the set-user-ID mode bit of the new process image file is 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 <code>setuid()</code> .	
9800	XSI Any shared memory segments attached to the calling process image shall not be attached to the new process image.	
9802	SEM Any named semaphores open in the calling process shall be closed as if by appropriate calls to <code>sem_close()</code> .	
9804	TYM Any blocks of typed memory that were mapped in the calling process are unmapped, as if <code>munmap()</code> was implicitly called to unmap them.	
9806	ML Memory locks established by the calling 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 the <code>exec</code> function. If the <code>exec</code> function fails, the effect on memory locks is unspecified.	
9811	MF SHM Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image.	
9813	When the calling process image does not use the <code>SCHED_FIFO</code> , <code>SCHED_RR</code> , or <code>SCHED_SPORADIC</code> scheduling policies, the scheduling policy and parameters of the new process image and the initial thread in that new process image are implementation-defined.	2
9814		2
9815		2
9816	PS When the calling process image uses the <code>SCHED_FIFO</code> , <code>SCHED_RR</code> , or <code>SCHED_SPORADIC</code> scheduling policies, the process policy and scheduling parameter settings shall not be changed by a call to an <code>exec</code> function. The initial thread in the new process image shall inherit the process scheduling policy and parameters. It shall have the default system contention scope, but shall inherit its allocation domain from the calling process image.	2
9818	TPS	2
9819		2
9820		2
9821	TMR Per-process timers created by the calling process shall be deleted before replacing the current process image with the new process image.	
9823	MSG All open message queue descriptors in the calling process shall be closed, as described in <code>mq_close()</code> .	
9825	AIO Any outstanding asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled shall complete as if the <code>exec</code> function had not yet occurred, but any associated signal notifications shall be suppressed. It is unspecified whether the <code>exec</code> function itself blocks awaiting such I/O completion. In no event, however, shall the new process image created by the <code>exec</code> function be affected by the presence of outstanding asynchronous I/O operations at the time the <code>exec</code> function is called. Whether any I/O is canceled, and which I/O may be canceled upon <code>exec</code> , is implementation-defined.	
9826		
9827		
9828		
9829		
9830		
9831		

9832	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 exec-ed shall not be reinitialized or altered as a result of the exec function other than to reflect the time spent by the process executing the exec function itself.	
9836	TCT	The initial value of the CPU-time clock of the initial thread of the new process image shall be set to zero.	
9838	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.	
9843		If the calling process is a trace controller process, any trace streams that were created by the calling process shall be shut down as described in the <i>posix_trace_shutdown()</i> function.	
9845	THR	The thread ID of the initial thread in the new process image is unspecified.	2
9846		The size and location of the stack on which the initial thread in the new process image runs is unspecified.	2
9848		The initial thread in the new process image shall have its cancellation type set to PTHREAD_CANCEL_DEFERRED and its cancellation state set to PTHREAD_CANCEL_ENABLED.	2
9851		The initial thread in the new process image shall have all thread-specific data values set to NULL and all thread-specific data keys shall be removed by the call to exec without running destructors.	2
9854		The initial thread in the new process image shall be joinable, as if created with the <i>detachstate</i> attribute set to PTHREAD_CREATE_JOINABLE.	2
9856		The new process shall inherit at least the following attributes from the calling process image:	
9857	XSI	<ul style="list-style-type: none"> • Nice value (see <i>nice()</i>) 	
9858	XSI	<ul style="list-style-type: none"> • <i>semadj</i> values (see <i>semop()</i>) 	
9859		<ul style="list-style-type: none"> • Process ID 	
9860		<ul style="list-style-type: none"> • Parent process ID 	
9861		<ul style="list-style-type: none"> • Process group ID 	
9862		<ul style="list-style-type: none"> • Session membership 	
9863		<ul style="list-style-type: none"> • Real user ID 	
9864		<ul style="list-style-type: none"> • Real group ID 	
9865		<ul style="list-style-type: none"> • Supplementary group IDs 	
9866		<ul style="list-style-type: none"> • Time left until an alarm clock signal (see <i>alarm()</i>) 	
9867		<ul style="list-style-type: none"> • Current working directory 	
9868		<ul style="list-style-type: none"> • Root directory 	
9869		<ul style="list-style-type: none"> • File mode creation mask (see <i>umask()</i>) 	
9870	XSI	<ul style="list-style-type: none"> • File size limit (see <i>ulimit()</i>) 	

9871		• Process signal mask (see <i>sigprocmask()</i>)	
9872		• Pending signal (see <i>sigpending()</i>)	
9873		• <i>tms_utime</i> , <i>tms_stime</i> , <i>tms_cutime</i> , and <i>tms_cstime</i> (see <i>times()</i>)	
9874 XSI		• Resource limits	
9875 XSI		• Controlling terminal	
9876 XSI		• Interval timers	
9877	The initial thread of the new process shall inherit at least the following attributes from the calling thread:		2
9878			2
9879	• Signal mask (see <i>sigprocmask()</i> and <i>pthread_sigmask()</i>)		2
9880	• Pending signals (see <i>sigpending()</i>)		2
9881	All other process attributes defined in this volume of IEEE Std 1003.1-2001 shall be inherited in the new process image from the old process image. All other thread attributes defined in this volume of IEEE Std 1003.1-2001 shall be inherited in the initial thread in the new process image from the calling thread in the old process image. The inheritance of process or thread attributes not defined by this volume of IEEE Std 1003.1-2001 is implementation-defined.		2
9882			2
9883			2
9884			2
9885			2
9886 THR	A call to any <i>exec</i> function from a process with more than one thread shall result in all threads being terminated and the new executable image being loaded and executed. No destructor functions or cleanup handlers shall be called.		2
9887			2
9888			2
9889	Upon successful completion, the <i>exec</i> functions shall mark for update the <i>st_atime</i> field of the file. If an <i>exec</i> function failed but was able to locate the process image file, whether the <i>st_atime</i> field is marked for update is unspecified. Should the <i>exec</i> function succeed, the process image file shall be considered to have been opened with <i>open()</i> . The corresponding <i>close()</i> shall be considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the <i>exec</i> functions, <i>posix_spawn()</i> , or <i>posix_spawnp()</i> . The <i>argv[]</i> and <i>envp[]</i> arrays of pointers and the strings to which those arrays point shall not be modified by a call to one of the <i>exec</i> functions, except as a consequence of replacing the process image.		2
9890			2
9891			2
9892			2
9893			2
9894			2
9895			2
9896			2
9897			2
9898 XSI	The saved resource limits in the new process image are set to be a copy of the process' corresponding hard and soft limits.		2
9899			2
9900	RETURN VALUE		
9901	If one of the <i>exec</i> functions returns to the calling process image, an error has occurred; the return value shall be -1 , and <i>errno</i> shall be set to indicate the error.		
9902			
9903	ERRORS		
9904	The <i>exec</i> functions shall fail if:		
9905	[E2BIG]	The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit of {ARG_MAX} bytes.	
9906			
9907			
9908	[EACCES]	Search permission is denied for a directory listed in the new process image file's path prefix, or the new process image file denies execution permission, or the new process image file is not a regular file and the implementation does not support execution of files of its type.	
9909			
9910			
9911			
9912	[EINVAL]	The new process image file has the appropriate permission and has a recognized executable binary format, but the system does not support execution of a file with this format.	
9913			
9914			

9915 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* or *file* argument.
9916
9917 [ENAMETOOLONG]
9918 The length of the *path* or *file* arguments exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
9919
9920 [ENOENT] A component of *path* or *file* does not name an existing file or *path* or *file* is an empty string.
9921
9922 [ENOTDIR] A component of the new process image file's path prefix is not a directory.
9923 The *exec* functions, except for *execlp()* and *execvp()*, shall fail if:
9924 [ENOEXEC] The new process image file has the appropriate access permission but has an unrecognized format.
9925
9926 The *exec* functions may fail if:
9927 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* or *file* argument.
9928
9929 [ENAMETOOLONG]
9930 As a result of encountering a symbolic link in resolution of the *path* argument,
9931 the length of the substituted pathname string exceeded {PATH_MAX}.
9932 [ENOMEM] The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.
9933
9934 [ETXTBSY] The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.
9935

9936 EXAMPLES

9937 Using *execl()*

9938 The following example executes the *ls* command, specifying the pathname of the executable
9939 (*/bin/ls*) and using arguments supplied directly to the command to produce single-column
9940 output.

```
9941 #include <unistd.h>
9942 int ret;
9943 ...
9944 ret = execl ("/bin/ls", "ls", "-l", (char *)0);
```

9945 Using *execle()*

9946 The following example is similar to **Using *execl()***. In addition, it specifies the environment for
9947 the new process image using the *env* argument.

```
9948 #include <unistd.h>
9949 int ret;
9950 char *env[ ] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
9951 ...
9952 ret = execle ("/bin/ls", "ls", "-l", (char *)0, env);
```

9953 **Using execp()**

9954 The following example searches for the location of the *ls* command among the directories
9955 specified by the *PATH* environment variable.

```
9956 #include <unistd.h>
9957 int ret;
9958 ...
9959 ret = execp ("ls", "ls", "-l", (char *)0);
```

9960 **Using execv()**

9961 The following example passes arguments to the *ls* command in the *cmd* array.

```
9962 #include <unistd.h>
9963 int ret;
9964 char *cmd[] = { "ls", "-l", (char *)0 };
9965 ...
9966 ret = execv ("/bin/ls", cmd);
```

9967 **Using execve()**

9968 The following example passes arguments to the *ls* command in the *cmd* array, and specifies the
9969 environment for the new process image using the *env* argument.

```
9970 #include <unistd.h>
9971 int ret;
9972 char *cmd[] = { "ls", "-l", (char *)0 };
9973 char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
9974 ...
9975 ret = execve ("/bin/ls", cmd, env);
```

9976 **Using execvp()**

9977 The following example searches for the location of the *ls* command among the directories
9978 specified by the *PATH* environment variable, and passes arguments to the *ls* command in the
9979 *cmd* array.

```
9980 #include <unistd.h>
9981 int ret;
9982 char *cmd[] = { "ls", "-l", (char *)0 };
9983 ...
9984 ret = execvp ("ls", cmd);
```

9985 APPLICATION USAGE

9986 As the state of conversion descriptors and message catalog descriptors in the new process image
9987 is undefined, conforming applications should not rely on their use and should close them prior
9988 to calling one of the *exec* functions.

9989 Applications that require other than the default POSIX locale should call *setlocale()* with the
9990 appropriate parameters to establish the locale of the new process.

9991 The *environ* array should not be accessed directly by the application.

9992 Applications should not depend on file descriptors 0, 1, and 2 being closed after an *exec*. A 1
9993 future version may allow these file descriptors to be automatically opened for any process. 1

9994 **RATIONALE**

9995 Early proposals required that the value of *argc* passed to *main()* be “one or greater”. This was
9996 driven by the same requirement in drafts of the ISO C standard. In fact, historical
9997 implementations have passed a value of zero when no arguments are supplied to the caller of
9998 the *exec* functions. This requirement was removed from the ISO C standard and subsequently
9999 removed from this volume of IEEE Std 1003.1-2001 as well. The wording, in particular the use of
10000 the word *should*, requires a Strictly Conforming POSIX Application to pass at least one argument
10001 to the *exec* function, thus guaranteeing that *argc* be one or greater when invoked by such an
10002 application. In fact, this is good practice, since many existing applications reference *argv[0]*
10003 without first checking the value of *argc*.

10004 The requirement on a Strictly Conforming POSIX Application also states that the value passed
10005 as the first argument be a filename associated with the process being started. Although some
10006 existing applications pass a pathname rather than a filename in some circumstances, a filename
10007 is more generally useful, since the common usage of *argv[0]* is in printing diagnostics. In some
10008 cases the filename passed is not the actual filename of the file; for example, many
10009 implementations of the *login* utility use a convention of prefixing a hyphen (‘-’) to the actual
10010 filename, which indicates to the command interpreter being invoked that it is a “login shell”.

10011 Historically there have been two ways that implementations can *exec* shell scripts.

10012 One common historical implementation is that the *execl()*, *execv()*, *execle()*, and *execve()*
10013 functions return an [ENOEXEC] error for any file not recognizable as executable, including a
10014 shell script. When the *execlp()* and *execvp()* functions encounter such a file, they assume the file
10015 to be a shell script and invoke a known command interpreter to interpret such files. This is now
10016 required by IEEE Std 1003.1-2001. These implementations of *execvp()* and *execlp()* only give the
10017 [ENOEXEC] error in the rare case of a problem with the command interpreter’s executable file.
10018 Because of these implementations, the [ENOEXEC] error is not mentioned for *execlp()* or
10019 *execvp()*, although implementations can still give it.

10020 Another way that some historical implementations handle shell scripts is by recognizing the first
10021 two bytes of the file as the character string "#!" and using the remainder of the first line of the
10022 file as the name of the command interpreter to execute.

10023 One potential source of confusion noted by the standard developers is over how the contents of
10024 a process image file affect the behavior of the *exec* family of functions. The following is a
10025 description of the actions taken:

- 10026 1. If the process image file is a valid executable (in a format that is executable and valid and
10027 having appropriate permission) for this system, then the system executes the file.
- 10028 2. If the process image file has appropriate permission and is in a format that is executable
10029 but not valid for this system (such as a recognized binary for another architecture), then
10030 this is an error and *errno* is set to [EINVAL] (see later RATIONALE on [EINVAL]).
- 10031 3. If the process image file has appropriate permission but is not otherwise recognized:
 - 10032 a. If this is a call to *execlp()* or *execvp()*, then they invoke a command interpreter
10033 assuming that the process image file is a shell script.
 - 10034 b. If this is not a call to *execlp()* or *execvp()*, then an error occurs and *errno* is set to
10035 [ENOEXEC].

10036 Applications that do not require to access their arguments may use the form:

10037 main(void)

10038 as specified in the ISO C standard. However, the implementation will always provide the two
10039 arguments *argc* and *argv*, even if they are not used.

10040 Some implementations provide a third argument to *main()* called *envp*. This is defined as a
10041 pointer to the environment. The ISO C standard specifies invoking *main()* with two arguments,
10042 so implementations must support applications written this way. Since this volume of
10043 IEEE Std 1003.1-2001 defines the global variable *environ*, which is also provided by historical
10044 implementations and can be used anywhere that *envp* could be used, there is no functional need
10045 for the *envp* argument. Applications should use the *getenv()* function rather than accessing the
10046 environment directly via either *envp* or *environ*. Implementations are required to support the
10047 two-argument calling sequence, but this does not prohibit an implementation from supporting
10048 *envp* as an optional third argument.

10049 This volume of IEEE Std 1003.1-2001 specifies that signals set to SIG_IGN remain set to
10050 SIG_IGN, and that the new process image inherits the signal mask of the thread that called *exec*
10051 in the old process image. This is consistent with historical implementations, and it permits some
10052 useful functionality, such as the *nohup* command. However, it should be noted that many
10053 existing applications wrongly assume that they start with certain signals set to the default action
10054 and/or unblocked. In particular, applications written with a simpler signal model that does not
10055 include blocking of signals, such as the one in the ISO C standard, may not behave properly if
10056 invoked with some signals blocked. Therefore, it is best not to block or ignore signals across
10057 *execs* without explicit reason to do so, and especially not to block signals across *execs* of arbitrary
10058 (not closely co-operating) programs.

2
2

10059 The *exec* functions always save the value of the effective user ID and effective group ID of the
10060 process at the completion of the *exec*, whether or not the set-user-ID or the set-group-ID bit of
10061 the process image file is set.

10062 The statement about *argv[]* and *envp[]* being constants is included to make explicit to future
10063 writers of language bindings that these objects are completely constant. Due to a limitation of
10064 the ISO C standard, it is not possible to state that idea in standard C. Specifying two levels of
10065 *const-qualification* for the *argv[]* and *envp[]* parameters for the *exec* functions may seem to be the
10066 natural choice, given that these functions do not modify either the array of pointers or the
10067 characters to which the function points, but this would disallow existing correct code. Instead,
10068 only the array of pointers is noted as constant. The table of assignment compatibility for *dst=src*
10069 derived from the ISO C standard summarizes the compatibility:

<i>dst:</i>	<i>char *[]</i>	<i>const char *[]</i>	<i>char *const[]</i>	<i>const char *const[]</i>
<i>src:</i>				
<i>char *[]</i>	VALID	—	VALID	—
<i>const char *[]</i>	—	VALID	—	VALID
<i>char * const []</i>	—	—	VALID	—
<i>const char *const[]</i>	—	—	—	VALID

10076 Since all existing code has a source type matching the first row, the column that gives the most
10077 valid combinations is the third column. The only other possibility is the fourth column, but
10078 using it would require a cast on the *argv* or *envp* arguments. It is unfortunate that the fourth
10079 column cannot be used, because the declaration a non-expert would naturally use would be that
10080 in the second row.

10081 The ISO C standard and this volume of IEEE Std 1003.1-2001 do not conflict on the use of
10082 *environ*, but some historical implementations of *environ* may cause a conflict. As long as *environ*
10083 is treated in the same way as an entry point (for example, *fork()*), it conforms to both standards.
10084 A library can contain *fork()*, but if there is a user-provided *fork()*, that *fork()* is given precedence

10085 and no problem ensues. The situation is similar for *environ*: the definition in this volume of
10086 IEEE Std 1003.1-2001 is to be used if there is no user-provided *environ* to take precedence. At
10087 least three implementations are known to exist that solve this problem.

10088 [E2BIG] The limit {ARG_MAX} applies not just to the size of the argument list, but to
10089 the sum of that and the size of the environment list.

10090 [EFAULT] Some historical systems return [EFAULT] rather than [ENOEXEC] when the
10091 new process image file is corrupted. They are non-conforming.

10092 [EINVAL] This error condition was added to IEEE Std 1003.1-2001 to allow an
10093 implementation to detect executable files generated for different architectures,
10094 and indicate this situation to the application. Historical implementations of
10095 shells, *execvp()*, and *execlp()* that encounter an [ENOEXEC] error will execute
10096 a shell on the assumption that the file is a shell script. This will not produce
10097 the desired effect when the file is a valid executable for a different
10098 architecture. An implementation may now choose to avoid this problem by
10099 returning [EINVAL] when a valid executable for a different architecture is
10100 encountered. Some historical implementations return [EINVAL] to indicate
10101 that the *path* argument contains a character with the high order bit set. The
10102 standard developers chose to deviate from historical practice for the following
10103 reasons:

10104 1. The new utilization of [EINVAL] will provide some measure of utility to
10105 the user community.

10106 2. Historical use of [EINVAL] is not acceptable in an internationalized
10107 operating environment.

10108 [ENAMETOOLONG] Since the file pathname may be constructed by taking elements in the *PATH*
10109 variable and putting them together with the filename, the
10110 [ENAMETOOLONG] error condition could also be reached this way.

10112 [ETXTBSY] System V returns this error when the executable file is currently open for
10113 writing by some process. This volume of IEEE Std 1003.1-2001 neither requires
10114 nor prohibits this behavior.

10115 Other systems (such as System V) may return [EINTR] from *exec*. This is not addressed by this
10116 volume of IEEE Std 1003.1-2001, but implementations may have a window between the call to
10117 *exec* and the time that a signal could cause one of the *exec* calls to return with [EINTR].

10118 An explicit statement regarding the floating-point environment (as defined in the <fenv.h>
10119 header) was added to make it clear that the floating-point environment is set to its default when
10120 a call to one of the *exec* functions succeeds. The requirements for inheritance or setting to the
10121 default for other process and thread start-up functions is covered by more generic statements in
10122 their descriptions and can be summarized as follows:

10123 *posix_spawn()* Set to default.

10124 *fork()* Inherit.

10125 *pthread_create()* Inherit.

10126 FUTURE DIRECTIONS

10127 None.

10128 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(), pthread_atfork(), pthread_sigmask(), 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>

2

10135 CHANGE HISTORY

10136 First released in Issue 1. Derived from Issue 1 of the SVID.

10137 Issue 5

10138 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

10140 Large File Summit extensions are added.

10141 Issue 6

10142 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- 10144 • In the DESCRIPTION, behavior is defined for when the process image file is not a valid executable.
- 10146 • 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.
- 10149 • The [ELOOP] mandatory error condition is added.
- 10150 • A second [ENAMETOOLONG] is added as an optional error condition.
- 10151 • The [ETXTBSY] optional error condition is added.

10152 The following changes were made to align with the IEEE P1003.1a draft standard:

- 10153 • The [EINVAL] mandatory error condition is added.
- 10154 • The [ELOOP] optional error condition is added.

10155 The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.

10156 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for typed memory.

10158 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

10159 The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

10160 IEEE PASC Interpretation 1003.1 #132 is applied.

10161 The DESCRIPTION is updated to make it explicit that the floating-point environment in the new process image is set to the default.

10163 The DESCRIPTION and RATIONALE are updated to include clarifications of how the contents of a process image file affect the behavior of the exec functions.

10165 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/15 is applied, adding a new paragraph to the DESCRIPTION and text to the end of the APPLICATION USAGE section. This change
10166 addresses a security concern, where implementations may want to reopen file descriptors 0, 1,
10167 and 2 for programs with the set-user-id or set-group-id file mode bits calling the exec family of
10168 functions.
10169

1
1
1
1

10170	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/24 is applied, applying changes to the DESCRIPTION, addressing which attributes are inherited by threads, and behavioral requirements for threads attributes.	2
10171		2
10172		2
10173	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/25 is applied, updating text in the RATIONALE from “the process signal mask be unchanged across an <i>exec</i> ” to “the new process image inherits the signal mask of the thread that called <i>exec</i> in the old process image”.	2
10174		2
10175		2

10176 NAME

10177 exit, _Exit, _exit — terminate a process

10178 SYNOPSIS

```
10179     #include <stdlib.h>
10180
10181     void exit(int status);
10182
10183     #include <unistd.h>
10184     void _exit(int status);
```

10184 DESCRIPTION

10185 CX For `exit()` and `_Exit()`: 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.

10188 CX The value of `status` may be 0, `EXIT_SUCCESS`, `EXIT_FAILURE`, or any other value, though only the least significant 8 bits (that is, `status & 0377`) shall be available to a waiting parent process.

10190 The `exit()` function shall first call all functions registered by `atexit()`, in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered. Each function is called as many times as it was registered. If, during the call to any such function, a call to the `longjmp()` function is made that would terminate the call to the registered function, the behavior is undefined.

10195 If a function registered by a call to `atexit()` fails to return, the remaining registered functions shall not be called and the rest of the `exit()` processing shall not be completed. If `exit()` is called more than once, the behavior is undefined.

10198 The `exit()` function shall then flush all open streams with unwritten buffered data, close all open streams, and remove all files created by `tmpfile()`. Finally, control shall be terminated with the consequences described below.

10201 CX The `_Exit()` and `_exit()` functions shall be functionally equivalent.

10202 CX The `_Exit()` and `_exit()` functions shall not call functions registered with `atexit()` nor any registered signal handlers. Whether open streams are flushed or closed, or temporary files are removed is implementation-defined. Finally, the calling process is terminated with the consequences described below.

10206 CX These functions shall terminate the calling process with the following consequences:

10207 Note: These consequences are all extensions to the ISO C standard and are not further CX shaded.
10208 However, XSI extensions are shaded.

- 10209 XSI • All of the file descriptors, directory streams, conversion descriptors, and message catalog descriptors open in the calling process shall be closed.

- 10211 XSI • If the parent process of the calling process is executing a `wait()` or `waitpid()`, and has neither set its `SA_NOCLDWAIT` flag nor set `SIGCHLD` to `SIG_IGN`, it shall be notified of the calling process' termination and the low-order eight bits (that is, bits 0377) of `status` shall be made available to it. If the parent is not waiting, the child's status shall be made available to it when the parent subsequently executes `wait()` or `waitpid()`.

10216 XSI The semantics of the `waitid()` function shall be equivalent to `wait()`.

- 10217 XSI • If the parent process of the calling process is not executing a `wait()` or `waitpid()`, and has neither set its `SA_NOCLDWAIT` flag nor set `SIGCHLD` to `SIG_IGN`, the calling process shall be transformed into a *zombie process*. A *zombie process* is an inactive process and it shall be

1
1

- 10220 deleted at some later time when its parent process executes `wait()` or `waitpid()`.
- 10221 XSI The semantics of the `waitid()` function shall be equivalent to `wait()`.
- 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.
 - Either:
- 10225 If the implementation supports the SIGCHLD signal, a SIGCHLD shall be sent to the parent process.
- 10227 Or:
- 10228 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.
- 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.
 - Each attached shared-memory segment is detached and the value of `shm_nattch` (see `shmget()`) in the data structure associated with its shared memory ID shall be decremented by 1.
 - For each semaphore for which the calling process has set a `semadj` value (see `semop()`), that value shall be added to the `semval` of the specified semaphore.
 - 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.
 - 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.
 - 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.
 - All open named semaphores in the calling process shall be closed as if by appropriate calls to `sem_close()`.
 - Any memory locks established by the process via calls to `mlockall()` or `mlock()` 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 `_Exit()` or `_exit()`.
 - Memory mappings that were created in the process shall be unmapped before the process is destroyed.
 - Any blocks of typed memory that were mapped in the calling process shall be unmapped, as if `munmap()` was implicitly called to unmap them.
 - All open message queue descriptors in the calling process shall be closed as if by appropriate calls to `mq_close()`.
 - Any outstanding cancelable asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled shall complete as if the `_Exit()` or `_exit()` operation had not yet occurred, but any associated signal notifications shall be suppressed.

10263 The `_Exit()` or `_exit()` operation may block awaiting such I/O completion. Whether any I/O
10264 is canceled, and which I/O may be canceled upon `_Exit()` or `_exit()`, is implementation-
10265 defined.

10266 • Threads terminated by a call to `_Exit()` or `_exit()` shall not invoke their cancellation cleanup
10267 handlers or per-thread data destructors.

10268 TRC • If the calling process is a trace controller process, any trace streams that were created by the
10269 calling process shall be shut down as described by the `posix_trace_shutdown()` function, and
10270 any process' mapping of trace event names to trace event type identifiers built for these trace
10271 streams may be deallocated.

10272 RETURN VALUE

10273 These functions do not return.

10274 ERRORS

10275 No errors are defined.

10276 EXAMPLES

10277 None.

10278 APPLICATION USAGE

10279 Normally applications should use `exit()` rather than `_Exit()` or `_exit()`.

10280 RATIONALE

10281 Process Termination

10282 Early proposals drew a distinction between normal and abnormal process termination.
10283 Abnormal termination was caused only by certain signals and resulted in implementation-
10284 defined "actions", as discussed below. Subsequent proposals distinguished three types of
10285 termination: *normal termination* (as in the current specification), *simple abnormal termination*, and
10286 *abnormal termination with actions*. Again the distinction between the two types of abnormal
10287 termination was that they were caused by different signals and that implementation-defined
10288 actions would result in the latter case. Given that these actions were completely
10289 implementation-defined, the early proposals were only saying when the actions could occur and
10290 how their occurrence could be detected, but not what they were. This was of little or no use to
10291 conforming applications, and thus the distinction is not made in this volume of
10292 IEEE Std 1003.1-2001.

10293 The implementation-defined actions usually include, in most historical implementations, the
10294 creation of a file named **core** in the current working directory of the process. This file contains an
10295 image of the memory of the process, together with descriptive information about the process,
10296 perhaps sufficient to reconstruct the state of the process at the receipt of the signal.

10297 There is a potential security problem in creating a **core** file if the process was set-user-ID and the
10298 current user is not the owner of the program, if the process was set-group-ID and none of the
10299 user's groups match the group of the program, or if the user does not have permission to write in
10300 the current directory. In this situation, an implementation either should not create a **core** file or
10301 should make it unreadable by the user.

10302 Despite the silence of this volume of IEEE Std 1003.1-2001 on this feature, applications are
10303 advised not to create files named **core** because of potential conflicts in many implementations.
10304 Some implementations use a name other than **core** for the file; for example, by appending the
10305 process ID to the filename.

10306

Terminating a Process

10307

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()`.

10311

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.

10316

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)`.

10319

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.

10325

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.

10327

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.

10336

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.

10349

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

10354 this volume of IEEE Std 1003.1-2001, but not in 4.2 BSD, stopped processes may be orphaned, but
10355 may be members of a process group that is not orphaned; therefore, the action taken at `_exit()`
10356 must consider processes other than child processes.

It is possible for a process group to be orphaned by a call to *setpgid()* or *setsid()*, as well as by process termination. This volume of IEEE Std 1003.1-2001 does not require sending SIGHUP and SIGCONT in those cases, because, unlike process termination, those cases are not caused accidentally by applications that are unaware of job control. An implementation can choose to send SIGHUP and SIGCONT in those cases as an extension; such an extension must be documented as required in <**signal.h**>.

10363 The ISO/IEC 9899:1999 standard adds the `_Exit()` function that results in immediate program
10364 termination without triggering signals or `atexit()`-registered functions. In IEEE Std 1003.1-2001,
10365 this is equivalent to the `_exit()` function.

10366 FUTURE DIRECTIONS

10367 None.

10368 SEE ALSO

10369 *atexit()*, *close()*, *fclose()*, *longjmp()*, *posix_trace_shutdown()*, *posix_trace_trid_eventid_open()*,
10370 *semop()*, *shmget()*, *sigaction()*, *wait()*, *waitid()*, *waitpid()*, the Base Definitions volume of
10371 IEEE Std 1003.1-2001, <stdlib.h>, <unistd.h>

10372 CHANGE HISTORY

10373 First released in Issue 1. Derived from Issue 1 of the SVID.

10374 Issue 5

10375 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
10376 Threads Extension.

10377 Interactions with the SA_NOCLDWAIT flag and SIGCHLD signal are further clarified.

10378 The values of *status* from `exit()` are better described.

10379 Issue 6

10380 Extensions beyond the ISO C standard are marked.

10381 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for
10382 typed memory.

10383 The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The `_Exit()` function is included.
 - The DESCRIPTION is updated.

10386 The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

10387 References to the `wait3()` function are removed.

10388 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/16 is applied, correcting grammar in the 1
10389 DESCRIPTION. 1

10390 NAME

10391 exp, expf, expl — exponential function

10392 SYNOPSIS

```
10393 #include <math.h>
10394 double exp(double x);
10395 float expf(float x);
10396 long double expl(long double x);
```

10397 DESCRIPTION

10398 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 10399 conflict between the requirements described here and the ISO C standard is unintentional. This
 10400 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

10401 These functions shall compute the base- e exponential of x .

10402 An application wishing to check for error situations should set *errno* to zero and call
 10403 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
 10404 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
 10405 zero, an error has occurred.

10406 RETURN VALUE

10407 Upon successful completion, these functions shall return the exponential value of x .

10408 If the correct value would cause overflow, a range error shall occur and *exp()*, *expf()*, and *expl()*
 10409 shall return the value of the macro `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`, respectively.

10410 If the correct value would cause underflow, and is not representable, a range error may occur,
 10411 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

10412 MX If x is NaN, a NaN shall be returned.

10413 If x is ± 0 , 1 shall be returned.

10414 If x is $-\infty$, $+0$ shall be returned.

10415 If x is $+\infty$, x shall be returned.

10416 If the correct value would cause underflow, and is representable, a range error may occur and
 10417 the correct value shall be returned.

10418 ERRORS

10419 These functions shall fail if:

10420 Range Error The result overflows.

10421 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 10422 then *errno* shall be set to [ERANGE]. If the integer expression
 10423 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 10424 floating-point exception shall be raised.

10425 These functions may fail if:

10426 Range Error The result underflows.

10427 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 10428 then *errno* shall be set to [ERANGE]. If the integer expression
 10429 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 10430 floating-point exception shall be raised.

10431 EXAMPLES

10432	Computing the Density of the Standard Normal Distribution	2
10433	This function shows an implementation for the density of the standard normal distribution	2
10434	using <code>exp()</code> . This example uses the constant M_PI which is an XSI extension.	2
10435	<code>#include <math.h></code>	2
10436	<code>double</code>	2
10437	<code>normal_density (double x)</code>	2
10438	<code>{</code>	2
10439	<code>return exp(-x*x/2) / sqrt (2*M_PI);</code>	2
10440	<code>}</code>	2

10441 APPLICATION USAGE

10442 Note that for IEEE Std 754-1985 `double`, $709.8 < x$ implies `exp(x)` has overflowed. The value
10443 $x < -708.4$ implies `exp(x)` has underflowed.

10444 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling &`
10445 `MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

10446 RATIONALE

10447 None.

10448 FUTURE DIRECTIONS

10449 None.

10450 SEE ALSO

10451 `feclearexcept()`, `fetestexcept()`, `isnan()`, `log()`, the Base Definitions volume of IEEE Std 1003.1-2001,
10452 Section 4.18, Treatment of Error Conditions for Mathematical Functions, `<math.h>`

10453 CHANGE HISTORY

10454 First released in Issue 1. Derived from Issue 1 of the SVID.

10455 Issue 5

10456 The DESCRIPTION is updated to indicate how an application should check for an error. This
10457 text was previously published in the APPLICATION USAGE section.

10458 Issue 6

10459 The `expf()` and `expl()` functions are added for alignment with the ISO/IEC 9899:1999 standard.

10460 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
10461 revised to align with the ISO/IEC 9899:1999 standard.

10462 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
10463 marked.

10464 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/26 is applied, adding the example to the
10465 EXAMPLES section. 2

10466 NAME

10467 exp2, exp2f, exp2l — exponential base 2 functions

10468 SYNOPSIS

```
10469     #include <math.h>
10470
10471     double exp2(double x);
10472     float exp2f(float x);
10473     long double exp2l(long double x);
```

10473 DESCRIPTION

10474 CX 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.

10477 These functions shall compute the base-2 exponential of x .

10478 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.

10482 RETURN VALUE

10483 Upon successful completion, these functions shall return 2^x .

10484 If the correct value would cause overflow, a range error shall occur and *exp2()*, *exp2f()*, and *exp2l()* shall return the value of the macro *HUGE_VAL*, *HUGE_VALF*, and *HUGE_VALL*, respectively.

10487 If the correct value would cause underflow, and is not representable, a range error may occur, and either 0.0 (if supported), or an implementation-defined value shall be returned.

10489 MX If x is NaN, a NaN shall be returned.

10490 If x is ± 0 , 1 shall be returned.

10491 If x is -Inf, +0 shall be returned.

10492 If x is +Inf, x shall be returned.

10493 If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

10495 ERRORS

10496 These functions shall fail if:

10497 Range Error The result overflows.

10498 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.

10502 These functions may fail if:

10503 Range Error The result underflows.

10504 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 underflow floating-point exception shall be raised.

10508 EXAMPLES

10509 None.

10510 APPLICATION USAGE

10511 For IEEE Std 754-1985 **double**, $1024 \leq x$ implies $\exp2(x)$ has overflowed. The value $x < -1022$ implies $\exp(x)$ has underflowed.

10513 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

10515 RATIONALE

10516 None.

10517 FUTURE DIRECTIONS

10518 None.

10519 SEE ALSO

10520 *exp()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *log()*, the Base Definitions volume of
10521 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
10522 *<math.h>*

10523 CHANGE HISTORY

10524 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

10525 **NAME**10526 `expm1, expm1f, expm1l — compute exponential functions`10527 **SYNOPSIS**

```
10528     #include <math.h>
10529
10530     double expm1(double x);
10531     float expm1f(float x);
10532     long double expm1l(long double x);
```

10532 **DESCRIPTION**

10533 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 10534 conflict between the requirements described here and the ISO C standard is unintentional. This
 10535 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

10536 These functions shall compute $e^x - 1.0$.

10537 An application wishing to check for error situations should set *errno* to zero and call
 10538 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
 10539 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
 10540 zero, an error has occurred.

10541 **RETURN VALUE**

10542 Upon successful completion, these functions return $e^x - 1.0$.

10543 If the correct value would cause overflow, a range error shall occur and *expm1()*, *expm1f()*, and
 10544 *expm1l()* shall return the value of the macro `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL`,
 10545 respectively.

10546 MX If *x* is NaN, a NaN shall be returned.

10547 If *x* is ± 0 , ± 0 shall be returned.

10548 If *x* is $-\infty$, -1 shall be returned.

10549 If *x* is ∞ , *x* shall be returned.

10550 If *x* is subnormal, a range error may occur and *x* should be returned.

10551 **ERRORS**

10552 These functions shall fail if:

10553 Range Error The result overflows.

10554 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 10555 then *errno* shall be set to [ERANGE]. If the integer expression
 10556 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 10557 floating-point exception shall be raised.

10558 These functions may fail if:

10559 MX Range Error The value of *x* is subnormal.

10560 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 10561 then *errno* shall be set to [ERANGE]. If the integer expression
 10562 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 10563 floating-point exception shall be raised.

10564 EXAMPLES

10565 None.

10566 APPLICATION USAGE

10567 The value of *expm1(x)* may be more accurate than *exp(x)–1.0* for small values of *x*.

10568 The *expm1()* and *log1p()* functions are useful for financial calculations of $((1+x)^n - 1)/x$, namely:

10569 $\expm1(n * \log1p(x)) / x$

10570 when *x* is very small (for example, when calculating small daily interest rates). These functions
10571 also simplify writing accurate inverse hyperbolic functions.

10572 For IEEE Std 754-1985 **double**, $709.8 < x$ implies *expm1(x)* has overflowed.

10573 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
10574 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

10575 RATIONALE

10576 None.

10577 FUTURE DIRECTIONS

10578 None.

10579 SEE ALSO

10580 *exp()*, *feclearexcept()*, *fetestexcept()*, *ilogb()*, *log1p()*, the Base Definitions volume of
10581 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
10582 **<math.h>**

10583 CHANGE HISTORY

10584 First released in Issue 4, Version 2.

10585 Issue 5

10586 Moved from X/OPEN UNIX extension to BASE.

10587 Issue 6

10588 The *expm1f()* and *expm1l()* functions are added for alignment with the ISO/IEC 9899:1999
10589 standard.

10590 The *expm1()* function is no longer marked as an extension.

10591 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
10592 revised to align with the ISO/IEC 9899:1999 standard.

10593 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
10594 marked.

10595 NAME

10596 **fabs, fabsf, fabsl** — absolute value function

10597 SYNOPSIS

```
10598       #include <math.h>
10599       double fabs(double x);
10600       float fabsf(float x);
10601       long double fabsl(long double x);
```

10602 DESCRIPTION

10603 CX The functionality described on this reference page is aligned with the ISO C standard. Any
10604 conflict between the requirements described here and the ISO C standard is unintentional. This
10605 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

10606 These functions shall compute the absolute value of their argument x , $|x|$.

10607 RETURN VALUE

10608 Upon successful completion, these functions shall return the absolute value of x .

10609 MX If x is NaN, a NaN shall be returned.

10610 If x is ± 0 , $+0$ shall be returned.

10611 If x is $\pm\text{Inf}$, $+\text{Inf}$ shall be returned.

10612 ERRORS

10613 No errors are defined.

10614 EXAMPLES**10615 Computing the 1-Norm of a Floating-Point Vector**

10616 This example shows the use of **fabs()** to compute the 1-norm of a vector defined as follows:

10617 $\text{norm1}(\mathbf{v}) = |\mathbf{v}[0]| + |\mathbf{v}[1]| + \dots + |\mathbf{v}[n-1]|$

10618 where $|x|$ denotes the absolute value of x , n denotes the vector's dimension $v[i]$ denotes the i -th
10619 component of \mathbf{v} ($0 \leq i < n$).

```
10620       #include <math.h>
10621       double
10622       norm1(const double v[], const int n)
10623       {
10624           int i;
10625           double n1_v; /* 1-norm of v */
10626           n1_v = 0;
10627           for (i=0; i<n; i++) {
10628               n1_v += fabs (v[i]);
10629           }
10630           return n1_v;
10631       }
```

10632 APPLICATION USAGE

10633 None.

10634 RATIONALE

10635 None.

10636 FUTURE DIRECTIONS

10637 None.

10638 SEE ALSO

10639 *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**math.h**>

10640 CHANGE HISTORY

10641 First released in Issue 1. Derived from Issue 1 of the SVID.

10642 Issue 5

10643 The DESCRIPTION is updated to indicate how an application should check for an error. This
10644 text was previously published in the APPLICATION USAGE section.

10645 Issue 6

10646 The *fabsf()* and *fabsl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

10647 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
10648 revised to align with the ISO/IEC 9899:1999 standard.

10649 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
10650 marked.

10651 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/27 is applied, adding the example to the 2
10652 EXAMPLES section. 2

10653 NAME

10654 fattach — attach a STREAMS-based file descriptor to a file in the file system name space
10655 (STREAMS)

10656 SYNOPSIS

10657 XSR #include <stropts.h>
10658 int fattach(int *fildes*, const char **path*);
10659

10660 DESCRIPTION

10661 The *fattach()* function shall attach a STREAMS-based file descriptor to a file, effectively
10662 associating a pathname with *fildes*. The application shall ensure that the *fildes* argument is a
10663 valid open file descriptor associated with a STREAMS file. The *path* argument points to a
10664 pathname of an existing file. The application shall have the appropriate privileges or be the
10665 owner of the file named by *path* and have write permission. A successful call to *fattach()* shall
10666 cause all pathnames that name the file named by *path* to name the STREAMS file associated with
10667 *fildes*, until the STREAMS file is detached from the file. A STREAMS file can be attached to more
10668 than one file and can have several pathnames associated with it.

10669 The attributes of the named STREAMS file shall be initialized as follows: the permissions, user
10670 ID, group ID, and times are set to those of the file named by *path*, the number of links is set to 1,
10671 and the size and device identifier are set to those of the STREAMS file associated with *fildes*. If
10672 any attributes of the named STREAMS file are subsequently changed (for example, by *chmod()*),
10673 neither the attributes of the underlying file nor the attributes of the STREAMS file to which *fildes*
10674 refers shall be affected.

10675 File descriptors referring to the underlying file, opened prior to an *fattach()* call, shall continue to
10676 refer to the underlying file.

10677 RETURN VALUE

10678 Upon successful completion, *fattach()* shall return 0. Otherwise, -1 shall be returned and *errno*
10679 set to indicate the error.

10680 ERRORS

10681 The *fattach()* function shall fail if:

- 10682 [EACCES] Search permission is denied for a component of the path prefix, or the process
10683 is the owner of *path* but does not have write permissions on the file named by
10684 *path*.
- 10685 [EBADF] The *fildes* argument is not a valid open file descriptor.
- 10686 [EBUSY] The file named by *path* is currently a mount point or has a STREAMS file
10687 attached to it.
- 10688 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
10689 argument.
- 10690 [ENAMETOOLONG] The size of *path* exceeds {PATH_MAX} or a component of *path* is longer than
10691 {NAME_MAX}.
- 10693 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.
- 10694 [ENOTDIR] A component of the path prefix is not a directory.
- 10695 [EPERM] The effective user ID of the process is not the owner of the file named by *path*
10696 and the process does not have appropriate privilege.

- 10697 The *fattach()* function may fail if:
- 10698 [EINVAL] The *fd* argument does not refer to a STREAMS file.
- 10699 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
10700 resolution of the *path* argument.
- 10701 [ENAMETOOLONG]
10702 Pathname resolution of a symbolic link produced an intermediate result
10703 whose length exceeds {PATH_MAX}.
- 10704 [EXDEV] A link to a file on another file system was attempted.

10705 EXAMPLES

10706 Attaching a File Descriptor to a File

10707 In the following example, *fd* refers to an open STREAMS file. The call to *fattach()* associates this
10708 STREAM with the file /tmp/named-STREAM, such that any future calls to open /tmp/named-
10709 STREAM, prior to breaking the attachment via a call to *fdetach()*, will instead create a new file
10710 handle referring to the STREAMS file associated with *fd*.

```
10711 #include <stropts.h>
10712 ...
10713     int fd;
10714     char *filename = "/tmp/named-STREAM";
10715     int ret;
10716
10717     ret = fattach(fd, filename);
```

10717 APPLICATION USAGE

10718 The *fattach()* function behaves similarly to the traditional *mount()* function in the way a file is
10719 temporarily replaced by the root directory of the mounted file system. In the case of *fattach()*, the
10720 replaced file need not be a directory and the replacing file is a STREAMS file.

10721 RATIONALE

10722 The file attributes of a file which has been the subject of an *fattach()* call are specifically set
10723 because of an artefact of the original implementation. The internal mechanism was the same as
10724 for the *mount()* function. Since *mount()* is typically only applied to directories, the effects when
10725 applied to a regular file are a little surprising, especially as regards the link count which rigidly
10726 remains one, even if there were several links originally and despite the fact that all original links
10727 refer to the STREAM as long as the *fattach()* remains in effect.

10728 FUTURE DIRECTIONS

10729 None.

10730 SEE ALSO

10731 *fdetach()*, *isastream()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stropts.h>

10732 CHANGE HISTORY

10733 First released in Issue 4, Version 2.

10734 Issue 5

10735 Moved from X/OPEN UNIX extension to BASE.

10736 The [EXDEV] error is added to the list of optional errors in the ERRORS section.

10737 **Issue 6**

- 10738 This function is marked as part of the XSI STREAMS Option Group.
- 10739 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
- 10740 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
10741 [ELOOP] error condition is added.

10742 NAME

10743 fchdir — change working directory

10744 SYNOPSIS

10745 XSI #include <unistd.h>

10746 int fchdir(int *fd*);

10747

10748 DESCRIPTION

10749 The *fchdir()* function shall be equivalent to *chdir()* except that the directory that is to be the new
10750 current working directory is specified by the file descriptor *fd*.

10751 A conforming application can obtain a file descriptor for a file of type directory using *open()*,
10752 provided that the file status flags and access modes do not contain O_WRONLY or O_RDWR.

10753 RETURN VALUE

10754 Upon successful completion, *fchdir()* shall return 0. Otherwise, it shall return -1 and set *errno* to
10755 indicate the error. On failure the current working directory shall remain unchanged.

10756 ERRORS

10757 The *fchdir()* function shall fail if:

10758 [EACCES] Search permission is denied for the directory referenced by *fd*.

10759 [EBADF] The *fd* argument is not an open file descriptor.

10760 [ENOTDIR] The open file descriptor *fd* does not refer to a directory.

10761 The *fchdir()* may fail if:

10762 [EINTR] A signal was caught during the execution of *fchdir()*.

10763 [EIO] An I/O error occurred while reading from or writing to the file system.

10764 EXAMPLES

10765 None.

10766 APPLICATION USAGE

10767 None.

10768 RATIONALE

10769 None.

10770 FUTURE DIRECTIONS

10771 None.

10772 SEE ALSO

10773 *chdir()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

10774 CHANGE HISTORY

10775 First released in Issue 4, Version 2.

10776 Issue 5

10777 Moved from X/OPEN UNIX extension to BASE.

10778 NAME

10779 fchmod — change mode of a file

10780 SYNOPSIS

```
10781 #include <sys/stat.h>
10782 int fchmod(int fildes, mode_t mode);
```

10783 DESCRIPTION

10784 The *fchmod()* function shall be equivalent to *chmod()* except that the file whose permissions are
10785 changed is specified by the file descriptor *fildes*.

10786 SHM If *fildes* references a shared memory object, the *fchmod()* function need only affect the S_IRUSR,
10787 S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits.

10788 TYM If *fildes* references a typed memory object, the behavior of *fchmod()* is unspecified.

10789 If *fildes* refers to a socket, the behavior of *fchmod()* is unspecified.

10790 XSR If *fildes* refers to a STREAM (which is *fattach()*-ed into the file system name space) the call
10791 returns successfully, doing nothing.

10792 RETURN VALUE

10793 Upon successful completion, *fchmod()* shall return 0. Otherwise, it shall return -1 and set *errno* to
10794 indicate the error.

10795 ERRORS

10796 The *fchmod()* function shall fail if:

10797 [EBADF] The *fildes* argument is not an open file descriptor.

10798 [EPERM] The effective user ID does not match the owner of the file and the process
10799 does not have appropriate privilege.

10800 [EROFS] The file referred to by *fildes* resides on a read-only file system.

10801 The *fchmod()* function may fail if:

10802 XSI [EINTR] The *fchmod()* function was interrupted by a signal.

10803 XSI [EINVAL] The value of the *mode* argument is invalid.

10804 [EINVAL] The *fildes* argument refers to a pipe and the implementation disallows
10805 execution of *fchmod()* on a pipe.

10806 EXAMPLES

10807 **Changing the Current Permissions for a File**

10808 The following example shows how to change the permissions for a file named /home/cnd/mod1
10809 so that the owner and group have read/write/execute permissions, but the world only has
10810 read/write permissions.

```
10811 #include <sys/stat.h>
10812 #include <fcntl.h>
10813 mode_t mode;
10814 int fildes;
10815 ...
10816 fildes = open( "/home/cnd/mod1" , O_RDWR );
10817 fchmod(fildes, S_IRWXU | S_IRWXG | S_IROTH | S_IWOTH );
```

10818 APPLICATION USAGE

10819 None.

10820 RATIONALE

10821 None.

10822 FUTURE DIRECTIONS

10823 None.

10824 SEE ALSO

10825 *chmod()*, *chown()*, *creat()*, *fcntl()*, *fstatvfs()*, *mknod()*, *open()*, *read()*, *stat()*, *write()*, the Base
10826 Definitions volume of IEEE Std 1003.1-2001, <sys/stat.h>

10827 CHANGE HISTORY

10828 First released in Issue 4, Version 2.

10829 Issue 5

10830 Moved from X/OPEN UNIX extension to BASE and aligned with *fchmod()* in the POSIX
10831 Realtime Extension. Specifically, the second paragraph of the DESCRIPTION is added and a
10832 second instance of [EINVAL] is defined in the list of optional errors.

10833 Issue 6

10834 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by stating that *fchmod()*
10835 behavior is unspecified for typed memory objects.

10836 NAME

10837 fchown — change owner and group of a file

10838 SYNOPSIS

```
10839 #include <unistd.h>
10840 int fchown(int fildes, uid_t owner, gid_t group);
```

10841 DESCRIPTION

10842 The *fchown()* function shall be equivalent to *chown()* except that the file whose owner and group
10843 are changed is specified by the file descriptor *fildes*.

10844 RETURN VALUE

10845 Upon successful completion, *fchown()* shall return 0. Otherwise, it shall return -1 and set *errno* to
10846 indicate the error.

10847 ERRORS

10848 The *fchown()* function shall fail if:

10849 [EBADF] The *fildes* argument is not an open file descriptor.

10850 [EPERM] The effective user ID does not match the owner of the file or the process does
10851 not have appropriate privilege and *_POSIX_CHOWN_RESTRICTED* indicates
10852 that such privilege is required.

10853 [EROFS] The file referred to by *fildes* resides on a read-only file system.

10854 The *fchown()* function may fail if:

10855 [EINVAL] The owner or group ID is not a value supported by the implementation. The
10856 XSR *fildes* argument refers to a pipe or socket or an *fattach()*-ed STREAM and the
10857 implementation disallows execution of *fchown()* on a pipe.

10858 [EIO] A physical I/O error has occurred.

10859 [EINTR] The *fchown()* function was interrupted by a signal which was caught.

10860 EXAMPLES

10861 Changing the Current Owner of a File

10862 The following example shows how to change the owner of a file named */home/cnd/mod1* to
10863 "jones" and the group to "cnd".

10864 The numeric value for the user ID is obtained by extracting the user ID from the user database
10865 entry associated with "jones". Similarly, the numeric value for the group ID is obtained by
10866 extracting the group ID from the group database entry associated with "cnd". This example
10867 assumes the calling program has appropriate privileges.

```
10868 #include <sys/types.h>
10869 #include <unistd.h>
10870 #include <fcntl.h>
10871 #include <pwd.h>
10872 #include <grp.h>

10873 struct passwd *pwd;
10874 struct group *grp;
10875 int fildes;
10876 ...
10877 fildes = open( "/home/cnd/mod1" , O_RDWR );
10878 pwd = getpwnam( "jones" );
```

```
10879     grp = getgrnam( "cnd" );
10880     fchown(fildes, pwd->pw_uid, grp->gr_gid);
10881 APPLICATION USAGE
10882     None.
10883 RATIONALE
10884     None.
10885 FUTURE DIRECTIONS
10886     None.
10887 SEE ALSO
10888     chown(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>
10889 CHANGE HISTORY
10890     First released in Issue 4, Version 2.
10891 Issue 5
10892     Moved from X/OPEN UNIX extension to BASE.
10893 Issue 6
10894     The following changes were made to align with the IEEE P1003.1a draft standard:
10895         • Clarification is added that a call to fchown() may not be allowed on a pipe.
10896     The fchown() function is defined as mandatory.
```

10897 NAME

10898 `fclose` — close a stream

10899 SYNOPSIS

10900 `#include <stdio.h>`

10901 `int fclose(FILE *stream);`

10902 DESCRIPTION

10903 CX 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.

10906 The `fclose()` function shall cause the stream pointed to by *stream* to be flushed and the associated file to be closed. Any unwritten buffered data for the stream shall be written to the file; any unread buffered data shall be discarded. Whether or not the call succeeds, the stream shall be disassociated from the file and any buffer set by the `setbuf()` or `setvbuf()` function shall be disassociated from the stream. If the associated buffer was automatically allocated, it shall be deallocated.

10912 CX The `fclose()` function shall mark for update the *st_ctime* and *st_mtime* fields of the underlying file, if the stream was writable, and if buffered data remains that has not yet been written to the file. The `fclose()` function shall perform the equivalent of a `close()` on the file descriptor that is associated with the stream pointed to by *stream*.

10916 After the call to `fclose()`, any use of *stream* results in undefined behavior.

10917 RETURN VALUE

10918 CX Upon successful completion, `fclose()` shall return 0; otherwise, it shall return EOF and set *errno* to indicate the error.

10920 ERRORS

10921 The `fclose()` function shall fail if:

10922 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the thread would be delayed in the write operation.

2

10924 CX [EBADF] The file descriptor underlying stream is not valid.

10925 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

10926 XSI [EFBIG] An attempt was made to write a file that exceeds the process' file size limit.

10927 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.

10929 CX [EINTR] The `fclose()` function was interrupted by a signal.

10930 CX [EIO] The process is a member of a background process group attempting to write 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.

10934 CX [ENOSPC] There was no free space remaining on the device containing the file.

10935 CX [EPIPE] An attempt is 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.

10937 The `fclose()` function may fail if:

10938 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

10940 EXAMPLES

10941 None.

10942 APPLICATION USAGE

10943 None.

10944 RATIONALE

10945 None.

10946 FUTURE DIRECTIONS

10947 None.

10948 SEE ALSO

10949 *close()*, *fopen()*, *getrlimit()*, *ulimit()*, the Base Definitions volume of IEEE Std 1003.1-2001,
10950 *<stdio.h>*

10951 CHANGE HISTORY

10952 First released in Issue 1. Derived from Issue 1 of the SVID.

10953 Issue 5

10954 Large File Summit extensions are added.

10955 Issue 6

10956 Extensions beyond the ISO C standard are marked.

10957 The following new requirements on POSIX implementations derive from alignment with the
10958 Single UNIX Specification:

- The [EFBIG] error is added as part of the large file support extensions.
- The [ENXIO] optional error condition is added.

10961 The DESCRIPTION is updated to note that the stream and any buffer are disassociated whether
10962 or not the call succeeds. This is for alignment with the ISO/IEC 9899:1999 standard.

10963 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/28 is applied, updating the [EAGAIN] 2
10964 error in the ERRORS section from “the process would be delayed” to “the thread would be 2
10965 delayed”. 2

10966 NAME

10967 fcntl — file control

10968 SYNOPSIS

```
10969 #include <unistd.h>
10970 #include <fcntl.h>
10971 int fcntl(int fd, int cmd, ...);
```

10972 DESCRIPTION

The *fcntl()* function shall perform the operations described below on open files. The *fd* argument is a file descriptor.

10975 The available values for *cmd* are defined in <**fcntl.h**> and are as follows:

10976 F_DUPFD	Return a new file descriptor which shall be the lowest numbered available (that is, not already open) file descriptor greater than or equal to the third argument, <i>arg</i> , taken as an integer of type int . The new file descriptor shall refer to the same open file description as the original file descriptor, and shall share any locks. The FD_CLOEXEC flag associated with the new file descriptor shall be cleared to keep the file open across calls to one of the <i>exec</i> functions.
10983 F_GETFD	Get the file descriptor flags defined in < fcntl.h > that are associated with the file descriptor <i>fd</i> . File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file.
10986 F_SETFD	Set the file descriptor flags defined in < fcntl.h >, that are associated with <i>fd</i> , to the third argument, <i>arg</i> , taken as type int . If the FD_CLOEXEC flag in the third argument is 0, the file shall remain open across the <i>exec</i> functions; otherwise, the file shall be closed upon successful execution of one of the <i>exec</i> functions.
10991 F_GETFL	Get the file status flags and file access modes, defined in < fcntl.h >, for the file description associated with <i>fd</i> . The file access modes can be extracted from the return value using the mask O_ACCMODE, which is defined in < fcntl.h >. File status flags and file access modes are associated with the file description and do not affect other file descriptors that refer to the same file with different open file descriptions.
10997 F_SETFL	Set the file status flags, defined in < fcntl.h >, for the file description associated with <i>fd</i> from the corresponding bits in the third argument, <i>arg</i> , taken as type int . Bits corresponding to the file access mode and the file creation flags, as defined in < fcntl.h >, that are set in <i>arg</i> shall be ignored. If any bits in <i>arg</i> other than those mentioned here are changed by the application, the result is unspecified.
11003 F_GETOWN	If <i>fd</i> refers to a socket, get the process or process group ID specified to receive SIGURG signals when out-of-band data is available. Positive values indicate a process ID; negative values, other than -1, indicate a process group ID. If <i>fd</i> does not refer to a socket, the results are unspecified.
11007 F_SETOWN	If <i>fd</i> refers to a socket, set the process or process group ID specified to receive SIGURG signals when out-of-band data is available, using the value of the third argument, <i>arg</i> , taken as type int . Positive values indicate a process ID; negative values, other than -1, indicate a process group ID. If <i>fd</i> does not refer to a socket, the results are unspecified.

11012 The following values for *cmd* are available for advisory record locking. Record locking shall be
11013 supported for regular files, and may be supported for other files.

11014 F_GETLK Get the first lock which blocks the lock description pointed to by the third
11015 argument, *arg*, taken as a pointer to type **struct flock**, defined in <fcntl.h>. The information
11016 retrieved shall overwrite the information passed to *fcntl()* in
11017 the structure **flock**. If no lock is found that would prevent this lock from
11018 being created, then the structure shall be left unchanged except for the lock
11019 type which shall be set to F_UNLCK.

11020 F_SETLK Set or clear a file segment lock according to the lock description pointed to by
11021 the third argument, *arg*, taken as a pointer to type **struct flock**, defined in
11022 <fcntl.h>. F_SETLK can establish shared (or read) locks (F_RDLCK) or
11023 exclusive (or write) locks (F_WRLCK), as well as to remove either type of lock
11024 (F_UNLCK). F_RDLCK, F_WRLCK, and F_UNLCK are defined in <fcntl.h>. If a shared or
11025 exclusive lock cannot be set, *fcntl()* shall return immediately
11026 with a return value of -1.

11027 F_SETLKW This command shall be equivalent to F_SETLK except that if a shared or
11028 exclusive lock is blocked by other locks, the thread shall wait until the request
11029 can be satisfied. If a signal that is to be caught is received while *fcntl()* is
11030 waiting for a region, *fcntl()* shall be interrupted. Upon return from the signal
11031 handler, *fcntl()* shall return -1 with *errno* set to [EINTR], and the lock
11032 operation shall not be done.

11033 Additional implementation-defined values for *cmd* may be defined in <fcntl.h>. Their names
11034 shall start with F_.

11035 When a shared lock is set on a segment of a file, other processes shall be able to set shared locks
11036 on that segment or a portion of it. A shared lock prevents any other process from setting an
11037 exclusive lock on any portion of the protected area. A request for a shared lock shall fail if the
11038 file descriptor was not opened with read access.

11039 An exclusive lock shall prevent any other process from setting a shared lock or an exclusive lock
11040 on any portion of the protected area. A request for an exclusive lock shall fail if the file
11041 descriptor was not opened with write access.

11042 The structure **flock** describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*),
11043 size (*l_len*), and process ID (*l_pid*) of the segment of the file to be affected.

11044 The value of *l_whence* is SEEK_SET, SEEK_CUR, or SEEK_END, to indicate that the relative
11045 offset *l_start* bytes shall be measured from the start of the file, current position, or end of the file,
11046 respectively. The value of *l_len* is the number of consecutive bytes to be locked. The value of
11047 *l_len* may be negative (where the definition of **off_t** permits negative values of *l_len*). The *l_pid*
11048 field is only used with F_GETLK to return the process ID of the process holding a blocking lock.
11049 After a successful F_GETLK request, when a blocking lock is found, the values returned in the
11050 **flock** structure shall be as follows:

11051 *l_type* Type of blocking lock found.
11052 *l_whence* SEEK_SET.
11053 *l_start* Start of the blocking lock.
11054 *l_len* Length of the blocking lock.
11055 *l_pid* Process ID of the process that holds the blocking lock.

If the command is F_SETLKW and the process must wait for another process to release a lock, then the range of bytes to be locked shall be determined before the *fcntl()* function blocks. If the file size or file descriptor seek offset change while *fcntl()* is blocked, this shall not affect the range of bytes locked.

If *l_len* is positive, the area affected shall start at *l_start* and end at *l_start+l_len*-1. If *l_len* is negative, the area affected shall start at *l_start+l_len* and end at *l_start*-1. Locks may start and extend beyond the current end of a file, but shall not extend before the beginning of the file. A lock shall be set to extend to the largest possible value of the file offset for that file by setting *l_len* to 0. If such a lock also has *l_start* set to 0 and *l_whence* is set to SEEK_SET, the whole file shall be locked.

There shall be at most one type of lock set for each byte in the file. Before a successful return from an F_SETLK or an F_SETLKW request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region shall be replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an F_SETLK or an F_SETLKW request (respectively) shall fail or block when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

All locks associated with a file for a given process shall be removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock another process' locked region. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, *fcntl()* shall fail with an [EDEADLK] error.

An unlock (F_UNLCK) request in which *l_len* is non-zero and the offset of the last byte of the requested segment is the maximum value for an object of type *off_t*, when the process has an existing lock in which *l_len* is 0 and which includes the last byte of the requested segment, shall be treated as a request to unlock from the start of the requested segment with an *l_len* equal to 0. Otherwise, an unlock (F_UNLCK) request shall attempt to unlock only the requested segment.

SHM When the file descriptor *fd* refers to a shared memory object, the behavior of *fcntl()* shall be the same as for a regular file except the effect of the following values for the argument *cmd* shall be unspecified: F_SETFL, F_GETLK, F_SETLK, and F_SETLKW.

TYM If *fd* refers to a typed memory object, the result of the *fcntl()* function is unspecified.

11088 RETURN VALUE

Upon successful completion, the value returned shall depend on *cmd* as follows:

11090	F_DUPFD	A new file descriptor.
11091	F_GETFD	Value of flags defined in <fcntl.h>. The return value shall not be negative.
11092	F_SETFD	Value other than -1.
11093	F_GETFL	Value of file status flags and access modes. The return value is not negative.
11094	F_SETFL	Value other than -1.
11095	F_GETLK	Value other than -1.
11096	F_SETLK	Value other than -1.
11097	F_SETLKW	Value other than -1.
11098	F_GETOWN	Value of the socket owner process or process group; this will not be -1.

11099	F_SETOWN	Value other than -1.	
11100		Otherwise, -1 shall be returned and <i>errno</i> set to indicate the error.	
11101	ERRORS		
11102	The <i>fcntl()</i> function shall fail if:		
11103	[EACCES] or [EAGAIN]		
11104		The <i>cmd</i> argument is F_SETLK; the type of lock (<i>l_type</i>) is a shared (F_RDLCK)	
11105		or exclusive (F_WRLCK) lock and the segment of a file to be locked is already	
11106		exclusive-locked by another process, or the type is an exclusive lock and some	
11107		portion of the segment of a file to be locked is already shared-locked or	
11108		exclusive-locked by another process.	
11109	[EBADF]	The <i>fildes</i> argument is not a valid open file descriptor, or the argument <i>cmd</i> is	
11110		F_SETLK or F_SETLKW, the type of lock, <i>l_type</i> , is a shared lock (F_RDLCK),	
11111		and <i>fildes</i> is not a valid file descriptor open for reading, or the type of lock,	
11112		<i>l_type</i> , is an exclusive lock (F_WRLCK), and <i>fildes</i> is not a valid file descriptor	
11113		open for writing.	
11114	[EINTR]	The <i>cmd</i> argument is F_SETLKW and the function was interrupted by a signal.	
11115	[EINVAL]	The <i>cmd</i> argument is invalid, or the <i>cmd</i> argument is F_DUPFD and <i>arg</i> is	
11116		negative or greater than or equal to {OPEN_MAX}, or the <i>cmd</i> argument is	
11117		F_GETLK, F_SETLK, or F_SETLKW and the data pointed to by <i>arg</i> is not valid,	
11118		or <i>fildes</i> refers to a file that does not support locking.	
11119	[EMFILE]	The argument <i>cmd</i> is F_DUPFD and {OPEN_MAX} file descriptors are	
11120		currently open in the calling process, or no file descriptors greater than or	
11121		equal to <i>arg</i> are available.	
11122	[ENOLCK]	The argument <i>cmd</i> is F_SETLK or F_SETLKW and satisfying the lock or unlock	
11123		request would result in the number of locked regions in the system exceeding	
11124		a system-imposed limit.	
11125	[EOVERFLOW]	One of the values to be returned cannot be represented correctly.	
11126	[EOVERFLOW]	The <i>cmd</i> argument is F_GETLK, F_SETLK, or F_SETLKW and the smallest or,	
11127		if <i>l_len</i> is non-zero, the largest offset of any byte in the requested segment	
11128		cannot be represented correctly in an object of type off_t .	
11129	The <i>fcntl()</i> function may fail if:		
11130	[EDEADLK]	The <i>cmd</i> argument is F_SETLKW, the lock is blocked by a lock from another	
11131		process, and putting the calling process to sleep to wait for that lock to	
11132		become free would cause a deadlock.	

11133 EXAMPLES

11134	Locking and Unlocking a File	2
11135	The following example demonstrates how to place a lock on bytes 100 to 109 of a file and then	2
11136	later remove it. F_SETLK is used to perform a non-blocking lock request so that the process does	2
11137	not have to wait if an incompatible lock is held by another process; instead the process can take	2
11138	some other action.	2
11139	#include <stdlib.h>	2
11140	#include <unistd.h>	2
11141	#include <fcntl.h>	2
11142	#include <errno.h>	2

```

11143     int
11144     main(int argc, char *argv[])
11145     {
11146         int fd;
11147         struct flock fl;
11148
11149         fd = open("testfile", O_RDWR);
11150         if (fd == -1)
11151             /* Handle error */;
11152
11153         /* Make a non-blocking request to place a write lock
11154          on bytes 100-109 of testfile */
11155
11156         fl.l_type = F_WRLCK;
11157         fl.l_whence = SEEK_SET;
11158         fl.l_start = 100;
11159         fl.l_len = 10;
11160
11161         if (fcntl(fd, F_SETLK, &fl) == -1) {
11162             if (errno == EACCES || errno == EAGAIN) {
11163                 printf("Already locked by another process\n");
11164
11165                 /* We can't get the lock at the moment */
11166
11167             } else {
11168                 /* Handle unexpected error */;
11169             }
11170         } else { /* Lock was granted... */
11171
11172             /* Perform I/O on bytes 100 to 109 of file */
11173
11174             /* Unlock the locked bytes */
11175
11176             fl.l_type = F_UNLCK;
11177             fl.l_whence = SEEK_SET;
11178             fl.l_start = 100;
11179             fl.l_len = 10;
11180
11181             if (fcntl(fd, F_SETLK, &fl) == -1)
11182                 /* Handle error */;
11183
11184         }
11185         exit(EXIT_SUCCESS);
11186     } /* main */

```

11176 Setting the Close-on-Exec Flag

11177 The following example demonstrates how to set the close-on-exec flag for the file descriptor *fd*.

```

11178 #include <unistd.h>
11179 #include <fcntl.h>
11180 ...
11181     int flags;
11182
11183     flags = fcntl(fd, F_GETFD);
11184     if (flags == -1)
11185         /* Handle error */;
11186     flags |= FD_CLOEXEC;
11187     if (fcntl(fd, F_SETFD, flags) == -1)
11188         /* Handle error */;"
```

11188 APPLICATION USAGE

11189 None.

11190 RATIONALE

11191 The ellipsis in the SYNOPSIS is the syntax specified by the ISO C standard for a variable number
11192 of arguments. It is used because System V uses pointers for the implementation of file locking
11193 functions.

11194 The *arg* values to F_GETFD, F_SETFD, F_GETFL, and F_SETFL all represent flag values to allow
11195 for future growth. Applications using these functions should do a read-modify-write operation
11196 on them, rather than assuming that only the values defined by this volume of
11197 IEEE Std 1003.1-2001 are valid. It is a common error to forget this, particularly in the case of
11198 F_SETFD.

11199 This volume of IEEE Std 1003.1-2001 permits concurrent read and write access to file data using
11200 the *fcntl()* function; this is a change from the 1984 /usr/group standard and early proposals.
11201 Without concurrency controls, this feature may not be fully utilized without occasional loss of
11202 data.

11203 Data losses occur in several ways. One case occurs when several processes try to update the
11204 same record, without sequencing controls; several updates may occur in parallel and the last
11205 writer “wins”. Another case is a bit-tree or other internal list-based database that is undergoing
11206 reorganization. Without exclusive use to the tree segment by the updating process, other reading
11207 processes chance getting lost in the database when the index blocks are split, condensed,
11208 inserted, or deleted. While *fcntl()* is useful for many applications, it is not intended to be overly
11209 general and does not handle the bit-tree example well.

11210 This facility is only required for regular files because it is not appropriate for many devices such
11211 as terminals and network connections.

11212 Since *fcntl()* works with “any file descriptor associated with that file, however it is obtained”,
11213 the file descriptor may have been inherited through a *fork()* or *exec* operation and thus may
11214 affect a file that another process also has open.

11215 The use of the open file description to identify what to lock requires extra calls and presents
11216 problems if several processes are sharing an open file description, but there are too many
11217 implementations of the existing mechanism for this volume of IEEE Std 1003.1-2001 to use
11218 different specifications.

11219 Another consequence of this model is that closing any file descriptor for a given file (whether or
11220 not it is the same open file description that created the lock) causes the locks on that file to be
11221 relinquished for that process. Equivalently, any close for any file/process pair relinquishes the
11222 locks owned on that file for that process. But note that while an open file description may be
11223 shared through *fork()*, locks are not inherited through *fork()*. Yet locks may be inherited through
11224 one of the *exec* functions.

11225 The identification of a machine in a network environment is outside the scope of this volume of
11226 IEEE Std 1003.1-2001. Thus, an *L_sysid* member, such as found in System V, is not included in the
11227 locking structure.

11228 Changing of lock types can result in a previously locked region being split into smaller regions.

11229 Mandatory locking was a major feature of the 1984 /usr/group standard.

11230 For advisory file record locking to be effective, all processes that have access to a file must
11231 cooperate and use the advisory mechanism before doing I/O on the file. Enforcement-mode
11232 record locking is important when it cannot be assumed that all processes are cooperating. For
11233 example, if one user uses an editor to update a file at the same time that a second user executes

another process that updates the same file and if only one of the two processes is using advisory locking, the processes are not cooperating. Enforcement-mode record locking would protect against accidental collisions.

Secondly, advisory record locking requires a process using locking to bracket each I/O operation with lock (or test) and unlock operations. With enforcement-mode file and record locking, a process can lock the file once and unlock when all I/O operations have been completed. Enforcement-mode record locking provides a base that can be enhanced; for example, with sharable locks. That is, the mechanism could be enhanced to allow a process to lock a file so other processes could read it, but none of them could write it.

Mandatory locks were omitted for several reasons:

1. Mandatory lock setting was done by multiplexing the set-group-ID bit in most implementations; this was confusing, at best.
2. The relationship to file truncation as supported in 4.2 BSD was not well specified.
3. Any publicly readable file could be locked by anyone. Many historical implementations keep the password database in a publicly readable file. A malicious user could thus prohibit logins. Another possibility would be to hold open a long-distance telephone line.
4. Some demand-paged historical implementations offer memory mapped files, and enforcement cannot be done on that type of file.

Since sleeping on a region is interrupted with any signal, *alarm()* may be used to provide a timeout facility in applications requiring it. This is useful in deadlock detection. Since implementation of full deadlock detection is not always feasible, the [EDEADLK] error was made optional.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm(), *close()*, *exec*, *open()*, *sigaction()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>, <signal.h>, <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

In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

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

- 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.
- In the DESCRIPTION, sentences describing behavior when *l_len* is negative are now mandated, and the description of unlock (F_UNLOCK) when *l_len* is non-negative is mandated.

- In the ERRORS section, the [EINVAL] error condition has the case mandated when the *cmd* is invalid, and two [EOVERFLOW] error conditions are added.

The F_GETOWN and F_SETOWN values are added for sockets.

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

- Clarification is added that the extent of the bytes locked is determined prior to the blocking action.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that `fcntl()` results are unspecified for typed memory objects.

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

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/29 is applied, adding the example to the EXAMPLES section.

11288 **NAME**11289 **fcvt** — convert a floating-point number to a string (**LEGACY**)11290 **SYNOPSIS**

```
11291 XSI #include <stdlib.h>
11292         char *fcvt(double value, int ndigit, int *restrict decpt,
11293                     int *restrict sign);
```

11294
11295 **DESCRIPTION**11296 Refer to **ecvt()**.

11297 NAME

11298 *fdatsync* — synchronize the data of a file (**REALTIME**)

11299 SYNOPSIS

11300 SIO `#include <unistd.h>`

11301 `int fdatsync(int fildes);`

11302

11303 DESCRIPTION

11304 The *fdatsync()* function shall force all currently queued I/O operations associated with the file indicated by file descriptor *fildes* to the synchronized I/O completion state.

11306 The functionality shall be equivalent to *fsync()* with the symbol `_POSIX_SYNCHRONIZED_IO` defined, with the exception that all I/O operations shall be completed as defined for synchronized I/O data integrity completion.

11309 RETURN VALUE

11310 If successful, the *fdatsync()* function shall return the value 0; otherwise, the function shall return the value -1 and set *errno* to indicate the error. If the *fdatsync()* function fails, outstanding I/O operations are not guaranteed to have been completed.

11313 ERRORS

11314 The *fdatsync()* function shall fail if:

11315 [EBADF] The *fildes* argument is not a valid file descriptor open for writing.

11316 [EINVAL] This implementation does not support synchronized I/O for this file.

11317 In the event that any of the queued I/O operations fail, *fdatsync()* shall return the error conditions defined for *read()* and *write()*.

11319 EXAMPLES

11320 None.

11321 APPLICATION USAGE

11322 None.

11323 RATIONALE

11324 None.

11325 FUTURE DIRECTIONS

11326 None.

11327 SEE ALSO

11328 *aio_fsync()*, *fcntl()*, *fsync()*, *open()*, *read()*, *write()*, the Base Definitions volume of
11329 IEEE Std 1003.1-2001, `<unistd.h>`

11330 CHANGE HISTORY

11331 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

11332 Issue 6

11333 The [ENOSYS] error condition has been removed as stubs need not be provided if an
11334 implementation does not support the Synchronized Input and Output option.

11335 The *fdatsync()* function is marked as part of the Synchronized Input and Output option.

11336 NAME

11337 fdetach — detach a name from a STREAMS-based file descriptor (**STREAMS**)

11338 SYNOPSIS

11339 XSR #include <stropts.h>

11340 int fdetach(const char *path);

11341

11342 DESCRIPTION

11343 The *fdetach()* function shall detach a STREAMS-based file from the file to which it was attached
11344 by a previous call to *fattach()*. The *path* argument points to the pathname of the attached
11345 STREAMS file. The process shall have appropriate privileges or be the owner of the file. A
11346 successful call to *fdetach()* shall cause all pathnames that named the attached STREAMS file to
11347 again name the file to which the STREAMS file was attached. All subsequent operations on *path*
11348 shall operate on the underlying file and not on the STREAMS file.

11349 All open file descriptions established while the STREAMS file was attached to the file referenced
11350 by *path* shall still refer to the STREAMS file after the *fdetach()* has taken effect.

11351 If there are no open file descriptors or other references to the STREAMS file, then a successful
11352 call to *fdetach()* shall be equivalent to performing the last *close()* on the attached file.

11353 RETURN VALUE

11354 Upon successful completion, *fdetach()* shall return 0; otherwise, it shall return -1 and set *errno* to
11355 indicate the error.

11356 ERRORS

11357 The *fdetach()* function shall fail if:

11358 [EACCES] Search permission is denied on a component of the path prefix.

11359 [EINVAL] The *path* argument names a file that is not currently attached.

11360 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
11361 argument.

11362 [ENAMETOOLONG]

11363 The size of a pathname exceeds {PATH_MAX} or a pathname component is
11364 longer than {NAME_MAX}.

11365 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

11366 [ENOTDIR] A component of the path prefix is not a directory.

11367 [EPERM] The effective user ID is not the owner of *path* and the process does not have
11368 appropriate privileges.

11369 The *fdetach()* function may fail if:

11370 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
11371 resolution of the *path* argument.

11372 [ENAMETOOLONG]

11373 Pathname resolution of a symbolic link produced an intermediate result
11374 whose length exceeds {PATH_MAX}.

11375 EXAMPLES**11376 Detaching a File**

11377 The following example detaches the STREAMS-based file **/tmp/named-STREAM** from the file to
11378 which it was attached by a previous, successful call to *fattach()*. Subsequent calls to open this
11379 file refer to the underlying file, not to the STREAMS file.

```
11380 #include <stropts.h>
11381 ...
11382     char *filename = "/tmp/named-STREAM";
11383     int ret;
11384
11384     ret = fdetach(filename);
```

11385 APPLICATION USAGE

11386 None.

11387 RATIONALE

11388 None.

11389 FUTURE DIRECTIONS

11390 None.

11391 SEE ALSO

11392 *fattach()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<stropts.h>**

11393 CHANGE HISTORY

11394 First released in Issue 4, Version 2.

11395 Issue 5

11396 Moved from X/OPEN UNIX extension to BASE.

11397 Issue 6

11398 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

11399 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
11400 [ELOOP] error condition is added.

11401 NAME

11402 *fdim*, *fdimf*, *fdiml* — compute positive difference between two floating-point numbers

11403 SYNOPSIS

```
11404     #include <math.h>
11405
11406     double fdim(double x, double y);
11407     float fdimf(float x, float y);
11408     long double fdiml(long double x, long double y);
```

11408 DESCRIPTION

11409 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11410 conflict between the requirements described here and the ISO C standard is unintentional. This
11411 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11412 These functions shall determine the positive difference between their arguments. If *x* is greater
11413 than *y*, *x*–*y* is returned. If *x* is less than or equal to *y*, +0 is returned.

11414 An application wishing to check for error situations should set *errno* to zero and call
11415 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
11416 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
11417 zero, an error has occurred.

11418 RETURN VALUE

11419 Upon successful completion, these functions shall return the positive difference value.

11420 If *x*–*y* is positive and overflows, a range error shall occur and *fdim()*, *fdimf()*, and *fdiml()* shall
11421 return the value of the macro *HUGE_VAL*, *HUGE_VALF*, and *HUGE_VALL*, respectively.

11422 XSI If *x*–*y* is positive and underflows, a range error may occur, and either (i) *(x*–*y*) (if representable), or
11423 0.0 (if supported), or an implementation-defined value shall be returned.

11424 MX If *x* or *y* is NaN, a NaN shall be returned.

11425 ERRORS

11426 The *fdim()* function shall fail if:

11427 Range Error The result overflows.

11428 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
11429 then *errno* shall be set to [ERANGE]. If the integer expression
11430 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
11431 floating-point exception shall be raised.

11432 The *fdim()* function may fail if:

11433 Range Error The result underflows.

11434 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
11435 then *errno* shall be set to [ERANGE]. If the integer expression
11436 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
11437 floating-point exception shall be raised.

11438 EXAMPLES

11439 None.

11440 APPLICATION USAGE

11441 On implementations supporting IEEE Std 754-1985, $x-y$ cannot underflow, and hence the 0.0 return value is shaded as an extension for implementations supporting the XSI extension rather than an MX extension.

11442
11443
11444 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling & MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

11446 RATIONALE

11447 None.

11448 FUTURE DIRECTIONS

11449 None.

11450 SEE ALSO

11451 *feclearexcept()*, *fetestexcept()*, *fmax()*, *fmin()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

11453 CHANGE HISTORY

11454 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11455 NAME

11456 fdopen — associate a stream with a file descriptor

11457 SYNOPSIS

11458 CX #include <stdio.h>

11459 FILE *fdopen(int *fildes*, const char **mode*);

11460

11461 DESCRIPTION

11462 The *fdopen()* function shall associate a stream with a file descriptor.

11463 The *mode* argument is a character string having one of the following values:

11464 *r* or *rb* Open a file for reading.

11465 *w* or *wb* Open a file for writing.

11466 *a* or *ab* Open a file for writing at end-of-file.

11467 *r+* or *rb+* or *r+b* Open a file for update (reading and writing).

11468 *w+* or *wb+* or *w+b* Open a file for update (reading and writing).

11469 *a+* or *ab+* or *a+b* Open a file for update (reading and writing) at end-of-file.

11470 The meaning of these flags is exactly as specified in *fopen()*, except that modes beginning with *w* shall not cause truncation of the file.

11472 Additional values for the *mode* argument may be supported by an implementation.

11473 The application shall ensure that the mode of the stream as expressed by the *mode* argument is allowed by the file access mode of the open file description to which *fildes* refers. The file position indicator associated with the new stream is set to the position indicated by the file offset associated with the file descriptor.

11477 The error and end-of-file indicators for the stream shall be cleared. The *fdopen()* function may cause the *st_atime* field of the underlying file to be marked for update.

11479 SHM If *fildes* refers to a shared memory object, the result of the *fdopen()* function is unspecified.

11480 TYP If *fildes* refers to a typed memory object, the result of the *fdopen()* function is unspecified.

11481 The *fdopen()* function shall preserve the offset maximum previously set for the open file description corresponding to *fildes*.

11483 RETURN VALUE

11484 Upon successful completion, *fdopen()* shall return a pointer to a stream; otherwise, a null pointer shall be returned and *errno* set to indicate the error.

11486 ERRORS

11487 The *fdopen()* function may fail if:

11488 [EBADF] The *fildes* argument is not a valid file descriptor.

11489 [EINVAL] The *mode* argument is not a valid mode.

11490 [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

11491 [EMFILE] {STREAM_MAX} streams are currently open in the calling process.

11492 [ENOMEM] Insufficient space to allocate a buffer.

11493 EXAMPLES

11494 None.

11495 APPLICATION USAGE

11496 File descriptors are obtained from calls like *open()*, *dup()*, *creat()*, or *pipe()*, which open files but
11497 do not return streams.

11498 RATIONALE

11499 The file descriptor may have been obtained from *open()*, *creat()*, *pipe()*, *dup()*, *fcntl()*, or *socket()*; 2
11500 inherited through *fork()*, *posix_spawn()*, or *exec*; or perhaps obtained by other means. 211501 The meanings of the *mode* arguments of *fdopen()* and *fopen()* differ. With *fdopen()*, open for write
11502 (*w* or *w+*) does not truncate, and append (*a* or *a+*) cannot create for writing. The *mode* argument
11503 formats that include *a b* are allowed for consistency with the ISO C standard function *fopen()*.
11504 The *b* has no effect on the resulting stream. Although not explicitly required by this volume of
11505 IEEE Std 1003.1-2001, a good implementation of append (*a*) mode would cause the O_APPEND
11506 flag to be set.

11507 FUTURE DIRECTIONS

11508 None.

11509 SEE ALSO

11510 Section 2.5.1 (on page 35), *fclose()*, *fopen()*, *open()*, *posix_spawn()*, *socket()*, the Base Definitions 2
11511 volume of IEEE Std 1003.1-2001, <stdio.h>

11512 CHANGE HISTORY

11513 First released in Issue 1. Derived from Issue 1 of the SVID.

11514 Issue 5

11515 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

11516 Large File Summit extensions are added.

11517 Issue 6

11518 The following new requirements on POSIX implementations derive from alignment with the
11519 Single UNIX Specification:

- 11520 • In the DESCRIPTION, the use and setting of the *mode* argument are changed to include
11521 binary streams.
- 11522 • In the DESCRIPTION, text is added for large file support to indicate setting of the offset
11523 maximum in the open file description.
- 11524 • All errors identified in the ERRORS section are added.
- 11525 • In the DESCRIPTION, text is added that the *fdopen()* function may cause *st_atime* to be
11526 updated.

11527 The following changes were made to align with the IEEE P1003.1a draft standard:

- 11528 • Clarification is added that it is the responsibility of the application to ensure that the mode is
11529 compatible with the open file descriptor.

11530 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
11531 *fdopen()* results are unspecified for typed memory objects.11532 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/30 is applied, making corrections to the 2
11533 RATIONALE. 2

11534 NAME

11535 *feclearexcept* — clear floating-point exception

11536 SYNOPSIS

11537 #include <fenv.h>

11538 int feclearexcept(int *excepts*);

11539 DESCRIPTION

11540 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11541 conflict between the requirements described here and the ISO C standard is unintentional. This
11542 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11543 The *feclearexcept()* function shall attempt to clear the supported floating-point exceptions
11544 represented by *excepts*.

11545 RETURN VALUE

11546 If the argument is zero or if all the specified exceptions were successfully cleared, *feclearexcept()*
11547 shall return zero. Otherwise, it shall return a non-zero value.

11548 ERRORS

11549 No errors are defined.

11550 EXAMPLES

11551 None.

11552 APPLICATION USAGE

11553 None.

11554 RATIONALE

11555 None.

11556 FUTURE DIRECTIONS

11557 None.

11558 SEE ALSO

11559 *fegetexceptflag()*, *feraiseexcept()*, *fesetexceptflag()*, *fetestexcept()*, the Base Definitions volume of
11560 IEEE Std 1003.1-2001, <fenv.h>

11561 CHANGE HISTORY

11562 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11563 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

11564 NAME

11565 fegetenv, fesetenv — get and set current floating-point environment

11566 SYNOPSIS

```
11567 #include <fenv.h>
11568 int fegetenv(fenv_t *envp);
11569 int fesetenv(const fenv_t *envp);
```

11570 DESCRIPTION

11571 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11572 conflict between the requirements described here and the ISO C standard is unintentional. This
11573 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11574 The *fegetenv()* function shall attempt to store the current floating-point environment in the object
11575 pointed to by *envp*.

11576 The *fesetenv()* function shall attempt to establish the floating-point environment represented by
11577 the object pointed to by *envp*. The argument *envp* shall point to an object set by a call to
11578 *fegetenv()* or *feholdexcept()*, or equal a floating-point environment macro. The *fesetenv()* function
11579 does not raise floating-point exceptions, but only installs the state of the floating-point status
11580 flags represented through its argument.

11581 RETURN VALUE

11582 If the representation was successfully stored, *fegetenv()* shall return zero. Otherwise, it shall
11583 return a non-zero value. If the environment was successfully established, *fesetenv()* shall return
11584 zero. Otherwise, it shall return a non-zero value.

11585 ERRORS

11586 No errors are defined.

11587 EXAMPLES

11588 None.

11589 APPLICATION USAGE

11590 None.

11591 RATIONALE

11592 None.

11593 FUTURE DIRECTIONS

11594 None.

11595 SEE ALSO

11596 *feholdexcept()*, *feupdateenv()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fenv.h>

11597 CHANGE HISTORY

11598 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11599 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

11600 NAME

11601 fegetexceptflag, fesetexceptflag — get and set floating-point status flags

11602 SYNOPSIS

```
11603 #include <fenv.h>
11604 int fegetexceptflag(fexcept_t *flagp, int excepts);
11605 int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

11606 DESCRIPTION

11607 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11608 conflict between the requirements described here and the ISO C standard is unintentional. This
11609 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11610 The *fegetexceptflag()* function shall attempt to store an implementation-defined representation of
11611 the states of the floating-point status flags indicated by the argument *excepts* in the object
11612 pointed to by the argument *flagp*.

11613 The *fesetexceptflag()* function shall attempt to set the floating-point status flags indicated by the
11614 argument *excepts* to the states stored in the object pointed to by *flagp*. The value pointed to by
11615 *flagp* shall have been set by a previous call to *fegetexceptflag()* whose second argument
11616 represented at least those floating-point exceptions represented by the argument *excepts*. This
11617 function does not raise floating-point exceptions, but only sets the state of the flags.

11618 RETURN VALUE

11619 If the representation was successfully stored, *fegetexceptflag()* shall return zero. Otherwise, it
11620 shall return a non-zero value. If the *excepts* argument is zero or if all the specified exceptions
11621 were successfully set, *fesetexceptflag()* shall return zero. Otherwise, it shall return a non-zero
11622 value.

11623 ERRORS

11624 No errors are defined.

11625 EXAMPLES

11626 None.

11627 APPLICATION USAGE

11628 None.

11629 RATIONALE

11630 None.

11631 FUTURE DIRECTIONS

11632 None.

11633 SEE ALSO

11634 *feclearexcept()*, *feraiseexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001,
11635 <fenv.h>

11636 CHANGE HISTORY

11637 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11638 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

11639 NAME

11640 *fgetround*, *fesetround* — get and set current rounding direction

11641 SYNOPSIS

```
11642        #include <fenv.h>
11643        int fegetround(void);
11644        int fesetround(int round);
```

11645 DESCRIPTION

11646 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11647 conflict between the requirements described here and the ISO C standard is unintentional. This
11648 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11649 The *fegetround()* function shall get the current rounding direction.

11650 The *fesetround()* function shall establish the rounding direction represented by its argument
11651 *round*. If the argument is not equal to the value of a rounding direction macro, the rounding
11652 direction is not changed.

11653 RETURN VALUE

11654 The *fegetround()* function shall return the value of the rounding direction macro representing the
11655 current rounding direction or a negative value if there is no such rounding direction macro or
11656 the current rounding direction is not determinable.

11657 The *fesetround()* function shall return a zero value if and only if the requested rounding direction
11658 was established.

11659 ERRORS

11660 No errors are defined.

11661 EXAMPLES

11662 The following example saves, sets, and restores the rounding direction, reporting an error and
11663 aborting if setting the rounding direction fails:

```
11664        #include <fenv.h>
11665        #include <assert.h>
11666        void f(int round_dir)
11667        {
11668           #pragma STDC FENV_ACCESS ON
11669           int save_round;
11670           int setround_ok;
11671           save_round = fegetround();
11672           setround_ok = fesetround(round_dir);
11673           assert(setround_ok == 0);
11674           /* ... */
11675           fesetround(save_round);
11676           /* ... */
11677        }
```

11678 APPLICATION USAGE

11679 None.

11680 RATIONALE

11681 None.

11682 FUTURE DIRECTIONS

11683 None.

11684 SEE ALSO

11685 The Base Definitions volume of IEEE Std 1003.1-2001, <fenv.h>

11686 CHANGE HISTORY

11687 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11688 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

11689 NAME

11690 *feholdexcept* — save current floating-point environment

11691 SYNOPSIS

11692 #include <fenv.h>

11693 int feholdexcept(fenv_t *envp);

11694 DESCRIPTION

11695 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11696 conflict between the requirements described here and the ISO C standard is unintentional. This
11697 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11698 The *feholdexcept()* function shall save the current floating-point environment in the object
11699 pointed to by *envp*, clear the floating-point status flags, and then install a non-stop (continue on
11700 floating-point exceptions) mode, if available, for all floating-point exceptions.

11701 RETURN VALUE

11702 The *feholdexcept()* function shall return zero if and only if non-stop floating-point exception
11703 handling was successfully installed.

11704 ERRORS

11705 No errors are defined.

11706 EXAMPLES

11707 None.

11708 APPLICATION USAGE

11709 None.

11710 RATIONALE

11711 The *feholdexcept()* function should be effective on typical IEC 60559:1989 standard
11712 implementations which have the default non-stop mode and at least one other mode for trap
11713 handling or aborting. If the implementation provides only the non-stop mode, then installing the
11714 non-stop mode is trivial.

11715 FUTURE DIRECTIONS

11716 None.

11717 SEE ALSO

11718 *fegetenv()*, *fesetenv()*, *feupdateenv()*, the Base Definitions volume of IEEE Std 1003.1-2001,
11719 <fenv.h>

11720 CHANGE HISTORY

11721 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11722 NAME

11723 *feof* — test end-of-file indicator on a stream

11724 SYNOPSIS

```
11725        #include <stdio.h>
11726        int feof(FILE *stream);
```

11727 DESCRIPTION

11728 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11729 conflict between the requirements described here and the ISO C standard is unintentional. This
11730 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11731 The *feof()* function shall test the end-of-file indicator for the stream pointed to by *stream*.

11732 RETURN VALUE

11733 The *feof()* function shall return non-zero if and only if the end-of-file indicator is set for *stream*.

11734 ERRORS

11735 No errors are defined.

11736 EXAMPLES

11737 None.

11738 APPLICATION USAGE

11739 None.

11740 RATIONALE

11741 None.

11742 FUTURE DIRECTIONS

11743 None.

11744 SEE ALSO

11745 *clearerr()*, *ferror()*, *fopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, *<stdio.h>*

11746 CHANGE HISTORY

11747 First released in Issue 1. Derived from Issue 1 of the SVID.

11748 NAME

11749 *feraiseexcept* — raise floating-point exception

11750 SYNOPSIS

```
11751        #include <fenv.h>
11752        int feraiseexcept(int excepts);
```

11753 DESCRIPTION

11754 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11755 conflict between the requirements described here and the ISO C standard is unintentional. This
11756 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11757 The *feraiseexcept()* function shall attempt to raise the supported floating-point exceptions
11758 represented by the argument *excepts*. The order in which these floating-point exceptions are
11759 raised is unspecified. Whether the *feraiseexcept()* function additionally raises the inexact
11760 floating-point exception whenever it raises the overflow or underflow floating-point exception is
11761 implementation-defined.

11762 RETURN VALUE

11763 If the argument is zero or if all the specified exceptions were successfully raised, *feraiseexcept()*
11764 shall return zero. Otherwise, it shall return a non-zero value.

11765 ERRORS

11766 No errors are defined.

11767 EXAMPLES

11768 None.

11769 APPLICATION USAGE

11770 The effect is intended to be similar to that of floating-point exceptions raised by arithmetic
11771 operations. Hence, enabled traps for floating-point exceptions raised by this function are taken.

11772 RATIONALE

11773 Raising overflow or underflow is allowed to also raise inexact because on some architectures the
11774 only practical way to raise an exception is to execute an instruction that has the exception as a
11775 side effect. The function is not restricted to accept only valid coincident expressions for atomic
11776 operations, so the function can be used to raise exceptions accrued over several operations.

11777 FUTURE DIRECTIONS

11778 None.

11779 SEE ALSO

11780 *feclearexcept()*, *fegetexceptflag()*, *fesetexceptflag()*, *fetestexcept()*, the Base Definitions volume of
11781 IEEE Std 1003.1-2001, <fenv.h>

11782 CHANGE HISTORY

11783 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11784 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

11785 NAME

11786 *ferror* — test error indicator on a stream

11787 SYNOPSIS

11788 #include <stdio.h>

11789 int ferror(FILE **stream*);

11790 DESCRIPTION

11791 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11792 conflict between the requirements described here and the ISO C standard is unintentional. This
11793 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11794 The *ferror()* function shall test the error indicator for the stream pointed to by *stream*.

11795 RETURN VALUE

11796 The *ferror()* function shall return non-zero if and only if the error indicator is set for *stream*.

11797 ERRORS

11798 No errors are defined.

11799 EXAMPLES

11800 None.

11801 APPLICATION USAGE

11802 None.

11803 RATIONALE

11804 None.

11805 FUTURE DIRECTIONS

11806 None.

11807 SEE ALSO

11808 *clearerr()*, *feof()*, *fopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

11809 CHANGE HISTORY

11810 First released in Issue 1. Derived from Issue 1 of the SVID.

11811 **NAME**
11812 **fesetenv** — set current floating-point environment
11813 **SYNOPSIS**
11814 #include <fenv.h>
11815 int fesetenv(const fenv_t *envp);
11816 **DESCRIPTION**
11817 Refer to *fegetenv()*.

11818 NAME

11819 `fesetexceptflag` — set floating-point status flags

11820 SYNOPSIS

11821 `#include <fenv.h>`

11822 `int fesetexceptflag(const fexcept_t *flagp, int excepts);`

11823 DESCRIPTION

11824 Refer to *fegetexceptflag()*.

11825 **NAME**

11826 fesetround — set current rounding direction

11827 **SYNOPSIS**

11828 #include <fenv.h>

11829 int fesetround(int round);

11830 **DESCRIPTION**11831 Refer to *fegetround()*.

11832 NAME

11833 *fetestexcept* — test floating-point exception flags

11834 SYNOPSIS

11835 #include <fenv.h>

11836 int fetestexcept(int *excepts*);

11837 DESCRIPTION

11838 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11839 conflict between the requirements described here and the ISO C standard is unintentional. This
11840 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11841 The *fetestexcept()* function shall determine which of a specified subset of the floating-point
11842 exception flags are currently set. The *excepts* argument specifies the floating-point status flags to
11843 be queried.

11844 RETURN VALUE

11845 The *fetestexcept()* function shall return the value of the bitwise-inclusive OR of the floating-point
11846 exception macros corresponding to the currently set floating-point exceptions included in
11847 *excepts*.

11848 ERRORS

11849 No errors are defined.

11850 EXAMPLES

11851 The following example calls function *f()* if an invalid exception is set, and then function *g()* if an
11852 overflow exception is set:

```
11853        #include <fenv.h>
11854        /* ... */
11855        {
11856            #pragma STDC FENV_ACCESS ON
11857            int set_excepts;
11858            feclearexcept(FE_INVALID | FE_OVERFLOW);
11859            // maybe raise exceptions
11860            set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
11861            if (set_excepts & FE_INVALID) f();
11862            if (set_excepts & FE_OVERFLOW) g();
11863            /* ... */
11864        }
```

11865 APPLICATION USAGE

11866 None.

11867 RATIONALE

11868 None.

11869 FUTURE DIRECTIONS

11870 None.

11871 SEE ALSO

11872 *feclearexcept()*, *fegetexceptflag()*, *feraiseexcept()*, the Base Definitions volume of
11873 IEEE Std 1003.1-2001, <fenv.h>

11874 CHANGE HISTORY

11875 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

11876 NAME

11877 feupdateenv — update floating-point environment

11878 SYNOPSIS

```
11879 #include <fenv.h>
11880 int feupdateenv(const fenv_t *envp);
```

11881 DESCRIPTION

11882 CX The functionality described on this reference page is aligned with the ISO C standard. Any
11883 conflict between the requirements described here and the ISO C standard is unintentional. This
11884 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11885 The *feupdateenv()* function shall attempt to save the currently raised floating-point exceptions in
11886 its automatic storage, attempt to install the floating-point environment represented by the object
11887 pointed to by *envp*, and then attempt to raise the saved floating-point exceptions. The argument
11888 *envp* shall point to an object set by a call to *feholdexcept()* or *fegetenv()*, or equal a floating-point
11889 environment macro.

11890 RETURN VALUE

11891 The *feupdateenv()* function shall return a zero value if and only if all the required actions were
11892 successfully carried out.

11893 ERRORS

11894 No errors are defined.

11895 EXAMPLES

11896 The following example shows sample code to hide spurious underflow floating-point
11897 exceptions:

```
11898 #include <fenv.h>
11899 double f(double x)
11900 {
11901     #pragma STDC FENV_ACCESS ON
11902     double result;
11903     fenv_t save_env;
11904     feholdexcept(&save_env);
11905     // compute result
11906     if /* test spurious underflow */
11907         feclearexcept(FE_UNDERFLOW);
11908     feupdateenv(&save_env);
11909     return result;
11910 }
```

11911 APPLICATION USAGE

11912 None.

11913 RATIONALE

11914 None.

11915 FUTURE DIRECTIONS

11916 None.

11917 SEE ALSO

11918 *fegetenv()*, *feholdexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fenv.h>

11919 CHANGE HISTORY

- 11920 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.
- 11921 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

11922 NAME

11923 **fflush** — flush a stream

11924 SYNOPSIS

```
11925        #include <stdio.h>
11926        int fflush(FILE *stream);
```

11927 DESCRIPTION

11928 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 11929 conflict between the requirements described here and the ISO C standard is unintentional. This
 11930 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

11931 If *stream* points to an output stream or an update stream in which the most recent operation was
 11932 CX not input, *fflush()* shall cause any unwritten data for that stream to be written to the file, and the
 11933 *st_ctime* and *st_mtime* fields of the underlying file shall be marked for update.

11934 If *stream* is a null pointer, *fflush()* shall perform this flushing action on all streams for which the
 11935 behavior is defined above.

11936 RETURN VALUE

11937 Upon successful completion, *fflush()* shall return 0; otherwise, it shall set the error indicator for
 11938 CX the stream, return EOF, and set *errno* to indicate the error.

11939 ERRORS

11940 The *fflush()* function shall fail if:

11941 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 11942 thread would be delayed in the write operation. 2

11943 CX [EBADF] The file descriptor underlying *stream* is not valid.

11944 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

11945 XSI [EFBIG] An attempt was made to write a file that exceeds the process' file size limit.

11946 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 11947 offset maximum associated with the corresponding stream.

11948 CX [EINTR] The *fflush()* function was interrupted by a signal.

11949 CX [EIO] The process is a member of a background process group attempting to write
 11950 to its controlling terminal, TOSTOP is set, the process is neither ignoring nor
 11951 blocking SIGTTOU, and the process group of the process is orphaned. This
 11952 error may also be returned under implementation-defined conditions.

11953 CX [ENOSPC] There was no free space remaining on the device containing the file.

11954 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
 11955 any process. A SIGPIPE signal shall also be sent to the thread.

11956 The *fflush()* function may fail if:

11957 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
 11958 capabilities of the device.

11959 EXAMPLES

11960 **Sending Prompts to Standard Output**

11961 The following example uses *printf()* calls to print a series of prompts for information the user
11962 must enter from standard input. The *fflush()* calls force the output to standard output. The
11963 *fflush()* function is used because standard output is usually buffered and the prompt may not
11964 immediately be printed on the output or terminal. The *gets()* calls read strings from standard
11965 input and place the results in variables, for use later in the program.

```
11966       #include <stdio.h>
11967       ...
11968       char user[100];
11969       char oldpasswd[100];
11970       char newpasswd[100];
11971       ...
11972       printf("User name: ");
11973       fflush(stdout);
11974       gets(user);

11975       printf("Old password: ");
11976       fflush(stdout);
11977       gets(oldpasswd);

11978       printf("New password: ");
11979       fflush(stdout);
11980       gets(newpasswd);
11981       ...
```

11982 APPLICATION USAGE

11983 None.

11984 RATIONALE

11985 Data buffered by the system may make determining the validity of the position of the current
11986 file descriptor impractical. Thus, enforcing the repositioning of the file descriptor after *flush()*
11987 on streams open for *read()* is not mandated by IEEE Std 1003.1-2001.

11988 FUTURE DIRECTIONS

11989 None.

11990 SEE ALSO

11991 *getrlimit()*, *ulimit()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

11992 CHANGE HISTORY

11993 First released in Issue 1. Derived from Issue 1 of the SVID.

11994 Issue 5

11995 Large File Summit extensions are added.

11996 Issue 6

11997 Extensions beyond the ISO C standard are marked.

11998 The following new requirements on POSIX implementations derive from alignment with the
11999 Single UNIX Specification:

- 12000 • The [EFBIG] error is added as part of the large file support extensions.
- 12001 • The [ENXIO] optional error condition is added.

12002	The RETURN VALUE section is updated to note that the error indicator shall be set for the stream. This is for alignment with the ISO/IEC 9899:1999 standard.	
12004	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/31 is applied, updating the [EAGAIN] error in the ERRORS section from “the process would be delayed” to “the thread would be delayed”.	2
12005		2
12006		2

12007 NAME

12008 *ffs* — find first set bit

12009 SYNOPSIS

12010 XSI

```
#include <strings.h>
```


12011

```
int ffs(int i);
```


12012

12013 DESCRIPTION

12014 The *ffs()* function shall find the first bit set (beginning with the least significant bit) in *i*, and
12015 return the index of that bit. Bits are numbered starting at one (the least significant bit).

12016 RETURN VALUE

12017 The *ffs()* function shall return the index of the first bit set. If *i* is 0, then *ffs()* shall return 0.

12018 ERRORS

12019 No errors are defined.

12020 EXAMPLES

12021 None.

12022 APPLICATION USAGE

12023 None.

12024 RATIONALE

12025 None.

12026 FUTURE DIRECTIONS

12027 None.

12028 SEE ALSO

12029 The Base Definitions volume of IEEE Std 1003.1-2001, *<strings.h>*

12030 CHANGE HISTORY

12031 First released in Issue 4, Version 2.

12032 Issue 5

12033 Moved from X/OPEN UNIX extension to BASE.

12034 NAME

12035 fgetc — get a byte from a stream

12036 SYNOPSIS

12037 #include <stdio.h>

12038 int fgetc(FILE *stream);

12039 DESCRIPTION

12040 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12041 conflict between the requirements described here and the ISO C standard is unintentional. This
 12042 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12043 If the end-of-file indicator for the input stream pointed to by *stream* is not set and a next byte is
 12044 present, the *fgetc()* function shall obtain the next byte as an **unsigned char** converted to an **int**,
 12045 from the input stream pointed to by *stream*, and advance the associated file position indicator for
 12046 the stream (if defined). Since *fgetc()* operates on bytes, reading a character consisting of multiple
 12047 bytes (or “a multi-byte character”) may require multiple calls to *fgetc()*.

12048 CX The *fgetc()* function may mark the *st_atime* field of the file associated with *stream* for update. The
 12049 *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*,
 12050 *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()*, or *scanf()* using *stream* that returns
 12051 data not supplied by a prior call to *ungetc()* or *ungetwc()*.

12052 RETURN VALUE

12053 Upon successful completion, *fgetc()* shall return the next byte from the input stream pointed to
 12054 by *stream*. If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the
 12055 end-of-file indicator for the stream shall be set and *fgetc()* shall return EOF. If a read error occurs,
 12056 CX the error indicator for the stream shall be set, *fgetc()* shall return EOF, and shall set *errno* to
 12057 indicate the error.

12058 ERRORS

12059 The *fgetc()* function shall fail if data needs to be read and:

12060 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 12061 thread would be delayed in the *fgetc()* operation.

2

12062 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
 12063 reading.

12064 CX [EINTR] The read operation was terminated due to the receipt of a signal, and no data
 12065 was transferred.

12066 CX [EIO] A physical I/O error has occurred, or the process is in a background process
 12067 group attempting to read from its controlling terminal, and either the process
 12068 is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
 12069 This error may also be generated for implementation-defined reasons.

12070 CX [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the
 12071 offset maximum associated with the corresponding stream.

12072 The *fgetc()* function may fail if:

12073 CX [ENOMEM] Insufficient storage space is available.

12074 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
 12075 capabilities of the device.

12076 EXAMPLES

12077 None.

12078 APPLICATION USAGE

12079 If the integer value returned by *fgetc()* is stored into a variable of type **char** and then compared against the integer constant EOF, the comparison may never succeed, because sign-extension of a variable of type **char** on widening to integer is implementation-defined.

12082 The *feof()* or *feof()* functions must be used to distinguish between an error condition and an end-of-file condition.

12084 RATIONALE

12085 None.

12086 FUTURE DIRECTIONS

12087 None.

12088 SEE ALSO

12089 *feof()*, *error()*, *fopen()*, *getchar()*, *getc()*, the Base Definitions volume of IEEE Std 1003.1-2001,
12090 <stdio.h>

12091 CHANGE HISTORY

12092 First released in Issue 1. Derived from Issue 1 of the SVID.

12093 Issue 5

12094 Large File Summit extensions are added.

12095 Issue 6

12096 Extensions beyond the ISO C standard are marked.

12097 The following new requirements on POSIX implementations derive from alignment with the
12098 Single UNIX Specification:

- 12099 • The [EIO] and [EOVERFLOW] mandatory error conditions are added.
- 12100 • The [ENOMEM] and [ENXIO] optional error conditions are added.

12101 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 12102 • The DESCRIPTION is updated to clarify the behavior when the end-of-file indicator for the
12103 input stream is not set.
- 12104 • The RETURN VALUE section is updated to note that the error indicator shall be set for the
12105 stream.

12106 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/32 is applied, updating the [EAGAIN] 2
12107 error in the ERRORS section from “the process would be delayed” to “the thread would be 2
12108 delayed”. 2

12109 NAME

12110 fgetpos — get current file position information

12111 SYNOPSIS

```
12112 #include <stdio.h>
12113 int fgetpos(FILE *restrict stream, fpos_t *restrict pos);
```

12114 DESCRIPTION

12115 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12116 conflict between the requirements described here and the ISO C standard is unintentional. This
12117 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12118 The *fgetpos()* function shall store the current values of the parse state (if any) and file position
12119 indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The value stored
12120 contains unspecified information usable by *fsetpos()* for repositioning the stream to its position
12121 at the time of the call to *fgetpos()*.

12122 RETURN VALUE

12123 Upon successful completion, *fgetpos()* shall return 0; otherwise, it shall return a non-zero value
12124 and set *errno* to indicate the error.

12125 ERRORS

12126 The *fgetpos()* function shall fail if:

12127 CX [EOVERFLOW] The current value of the file position cannot be represented correctly in an
12128 object of type **fpos_t**.

12129 The *fgetpos()* function may fail if:

12130 CX [EBADF] The file descriptor underlying *stream* is not valid.

12131 CX [ESPIPE] The file descriptor underlying *stream* is associated with a pipe, FIFO, or socket.
12132

12133 EXAMPLES

12134 None.

12135 APPLICATION USAGE

12136 None.

12137 RATIONALE

12138 None.

12139 FUTURE DIRECTIONS

12140 None.

12141 SEE ALSO

12142 *open()*, *ftell()*, *rewind()*, *ungetc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**stdio.h**>

12143 CHANGE HISTORY

12144 First released in Issue 4. Derived from the ISO C standard.

12145 Issue 5

12146 Large File Summit extensions are added.

12147 Issue 6

12148 Extensions beyond the ISO C standard are marked.

12149 The following new requirements on POSIX implementations derive from alignment with the
12150 Single UNIX Specification:

- 12151 • The [EBADF] and [ESPIPE] optional error conditions are added.
- 12152 An additional [ESPIPE] error condition is added for sockets.
- 12153 The prototype for *fgetpos()* is changed for alignment with the ISO/IEC 9899:1999 standard.

12154 NAME

12155 fgets — get a string from a stream

12156 SYNOPSIS

```
12157        #include <stdio.h>
12158        char *fgets(char *restrict s, int n, FILE *restrict stream);
```

12159 DESCRIPTION

12160 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12161 conflict between the requirements described here and the ISO C standard is unintentional. This
12162 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12163 The *fgets()* function shall read bytes from *stream* into the array pointed to by *s*, until *n*-1 bytes
12164 are read, or a <newline> is read and transferred to *s*, or an end-of-file condition is encountered.
12165 The string is then terminated with a null byte.

12166 CX The *fgets()* function may mark the *st_atime* field of the file associated with *stream* for update. The
12167 *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*,
12168 *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()*, or *scanf()* using *stream* that returns
12169 data not supplied by a prior call to *ungetc()* or *ungetwc()*.

12170 RETURN VALUE

12171 Upon successful completion, *fgets()* shall return *s*. If the stream is at end-of-file, the end-of-file
12172 indicator for the stream shall be set and *fgets()* shall return a null pointer. If a read error occurs,
12173 CX the error indicator for the stream shall be set, *fgets()* shall return a null pointer, and shall set
12174 *errno* to indicate the error.

12175 ERRORS

12176 Refer to *fgetc()*.

12177 EXAMPLES**12178 Reading Input**

12179 The following example uses *fgets()* to read each line of input. {LINE_MAX}, which defines the
12180 maximum size of the input line, is defined in the <limits.h> header.

```
12181        #include <stdio.h>
12182        ...
12183        char line[LINE_MAX];
12184        ...
12185        while (fgets(line, LINE_MAX, fp) != NULL) {
12186           ...
12187        }
12188        ...
```

12189 APPLICATION USAGE

12190 None.

12191 RATIONALE

12192 None.

12193 FUTURE DIRECTIONS

12194 None.

12195 SEE ALSO

12196 *fopen()*, *fread()*, *gets()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

12197 CHANGE HISTORY

12198 First released in Issue 1. Derived from Issue 1 of the SVID.

12199 Issue 6

12200 Extensions beyond the ISO C standard are marked.

12201 The prototype for *fgets()* is changed for alignment with the ISO/IEC 9899: 1999 standard.

12202 NAME

12203 fgetwc — get a wide-character code from a stream

12204 SYNOPSIS

```
12205 #include <stdio.h>
12206 #include <wchar.h>
12207 wint_t fgetwc(FILE *stream);
```

12208 DESCRIPTION

12209 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12210 conflict between the requirements described here and the ISO C standard is unintentional. This
 12211 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12212 The *fgetwc()* function shall obtain the next character (if present) from the input stream pointed to
 12213 by *stream*, convert that to the corresponding wide-character code, and advance the associated
 12214 file position indicator for the stream (if defined).

12215 If an error occurs, the resulting value of the file position indicator for the stream is unspecified.

12216 CX The *fgetwc()* function may mark the *st_atime* field of the file associated with *stream* for update.
 12217 The *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*,
 12218 *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()*, or *scanf()* using *stream* that returns
 12219 data not supplied by a prior call to *ungetc()* or *ungetwc()*.

12220 RETURN VALUE

12221 Upon successful completion, the *fgetwc()* function shall return the wide-character code of the
 12222 character read from the input stream pointed to by *stream* converted to a type **wint_t**. If the
 12223 stream is at end-of-file, the end-of-file indicator for the stream shall be set and *fgetwc()* shall
 12224 return WEOF. If a read error occurs, the error indicator for the stream shall be set, *fgetwc()* shall
 12225 CX return WEOF, and shall set *errno* to indicate the error. If an encoding error occurs, the error
 12226 indicator for the stream shall be set, *fgetwc()* shall return WEOF, and shall set *errno* to indicate
 12227 the error.

12228 ERRORS

12229 The *fgetwc()* function shall fail if data needs to be read and:

12230 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 12231 thread would be delayed in the *fgetwc()* operation.

2

12232 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
 12233 reading.

12234 [EILSEQ] The data obtained from the input stream does not form a valid character.

12235 CX [EINTR] The read operation was terminated due to the receipt of a signal, and no data
 12236 was transferred.

12237 CX [EIO] A physical I/O error has occurred, or the process is in a background process
 12238 group attempting to read from its controlling terminal, and either the process
 12239 is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
 12240 This error may also be generated for implementation-defined reasons.

12241 CX [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the
 12242 offset maximum associated with the corresponding stream.

12243 The *fgetwc()* function may fail if:

12244 CX [ENOMEM] Insufficient storage space is available.

12245 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
12246 capabilities of the device.

12247 EXAMPLES

12248 None.

12249 APPLICATION USAGE

12250 The *ferror()* or *feof()* functions must be used to distinguish between an error condition and an
12251 end-of-file condition.

12252 RATIONALE

12253 None.

12254 FUTURE DIRECTIONS

12255 None.

12256 SEE ALSO

12257 *feof()*, *ferror()*, *fopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<stdio.h>**,
12258 **<wchar.h>**

12259 CHANGE HISTORY

12260 First released in Issue 4. Derived from the MSE working draft.

12261 Issue 5

12262 The Optional Header (OH) marking is removed from **<stdio.h>**.

12263 Large File Summit extensions are added.

12264 Issue 6

12265 Extensions beyond the ISO C standard are marked.

12266 The following new requirements on POSIX implementations derive from alignment with the
12267 Single UNIX Specification:

- 12268 • The [EIO] and [EOVERFLOW] mandatory error conditions are added.
- 12269 • The [ENOMEM] and [ENXIO] optional error conditions are added.

12270 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/33 is applied, updating the [EAGAIN] 2
12271 error in the ERRORS section from “the process would be delayed” to “the thread would be 2
12272 delayed”. 2

12273 NAME

12274 *fgetws* — get a wide-character string from a stream

12275 SYNOPSIS

```
12276     #include <stdio.h>
12277     #include <wchar.h>
12278
12279     wchar_t *fgetws(wchar_t *restrict ws, int n,
12280                       FILE *restrict stream);
```

12280 DESCRIPTION

12281 CX 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.

12284 The *fgetws()* function shall read characters from the *stream*, convert these to the corresponding wide-character codes, place them in the *wchar_t* array pointed to by *ws*, until *n*-1 characters are read, or a <newline> is read, converted, and transferred to *ws*, or an end-of-file condition is encountered. The wide-character string, *ws*, shall then be terminated with a null wide-character code.

12289 If an error occurs, the resulting value of the file position indicator for the stream is unspecified.

12290 CX The *fgetws()* function may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*, *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()*, or *scanf()* using *stream* that returns data not supplied by a prior call to *ungetc()* or *ungetwc()*.

12294 RETURN VALUE

12295 Upon successful completion, *fgetws()* shall return *ws*. If the stream is at end-of-file, the end-of-file indicator for the stream shall be set and *fgetws()* shall return a null pointer. If a read error occurs, the error indicator for the stream shall be set, *fgetws()* shall return a null pointer, and shall set *errno* to indicate the error.

12299 ERRORS

12300 Refer to *fgetwc()*.

12301 EXAMPLES

12302 None.

12303 APPLICATION USAGE

12304 None.

12305 RATIONALE

12306 None.

12307 FUTURE DIRECTIONS

12308 None.

12309 SEE ALSO

12310 *open()*, *fread()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>, <wchar.h>

12311 CHANGE HISTORY

12312 First released in Issue 4. Derived from the MSE working draft.

12313 Issue 5

12314 The Optional Header (OH) marking is removed from <stdio.h>.

12315 Issue 6

- 12316 Extensions beyond the ISO C standard are marked.
- 12317 The prototype for *fgetws()* is changed for alignment with the ISO/IEC 9899:1999 standard.

12318 NAME

12319 *fileno* — map a stream pointer to a file descriptor

12320 SYNOPSIS

12321 CX #include <stdio.h>

12322 int fileno(FILE *stream);

12323

12324 DESCRIPTION

12325 The *fileno()* function shall return the integer file descriptor associated with the stream pointed to by *stream*.

12327 RETURN VALUE

12328 Upon successful completion, *fileno()* shall return the integer value of the file descriptor associated with *stream*. Otherwise, the value -1 shall be returned and *errno* set to indicate the error.

12331 ERRORS

12332 The *fileno()* function may fail if:

12333 [EBADF] The *stream* argument is not a valid stream.

12334 EXAMPLES

12335 None.

12336 APPLICATION USAGE

12337 None.

12338 RATIONALE

12339 Without some specification of which file descriptors are associated with these streams, it is impossible for an application to set up the streams for another application it starts with *fork()* and *exec*. In particular, it would not be possible to write a portable version of the *sh* command interpreter (although there may be other constraints that would prevent that portability).

12343 FUTURE DIRECTIONS

12344 None.

12345 SEE ALSO

12346 Section 2.5.1 (on page 35), *fdopen()*, *fopen()*, *stdin*, the Base Definitions volume of
12347 IEEE Std 1003.1-2001, <stdio.h>

12348 CHANGE HISTORY

12349 First released in Issue 1. Derived from Issue 1 of the SVID.

12350 Issue 6

12351 The following new requirements on POSIX implementations derive from alignment with the
12352 Single UNIX Specification:

- 12353 • The [EBADF] optional error condition is added.

12354 NAME

12355 flockfile, ftrylockfile, funlockfile — stdio locking functions

12356 SYNOPSIS

12357 TSF #include <stdio.h>
12358 void flockfile(FILE *file);
12359 int ftrylockfile(FILE *file);
12360 void funlockfile(FILE *file);
12361

12362 DESCRIPTION

12363 These functions shall provide for explicit application-level locking of stdio (**FILE ***) objects.
12364 These functions can be used by a thread to delineate a sequence of I/O statements that are
12365 executed as a unit.

12366 The *flockfile()* function shall acquire for a thread ownership of a (**FILE ***) object.

12367 The *ftrylockfile()* function shall acquire for a thread ownership of a (**FILE ***) object if the object is
12368 available; *ftrylockfile()* is a non-blocking version of *flockfile()*.

12369 The *funlockfile()* function shall relinquish the ownership granted to the thread. The behavior is
12370 undefined if a thread other than the current owner calls the *funlockfile()* function.

12371 The functions shall behave as if there is a lock count associated with each (**FILE ***) object. This
12372 count is implicitly initialized to zero when the (**FILE ***) object is created. The (**FILE ***) object is
12373 unlocked when the count is zero. When the count is positive, a single thread owns the (**FILE ***)
12374 object. When the *flockfile()* function is called, if the count is zero or if the count is positive and
12375 the caller owns the (**FILE ***) object, the count shall be incremented. Otherwise, the calling thread
12376 shall be suspended, waiting for the count to return to zero. Each call to *funlockfile()* shall
12377 decrement the count. This allows matching calls to *flockfile()* (or successful calls to *ftrylockfile()*)
12378 and *funlockfile()* to be nested.

12379 All functions that reference (**FILE ***) objects shall behave as if they use *flockfile()* and *funlockfile()*
12380 internally to obtain ownership of these (**FILE ***) objects.

12381 RETURN VALUE

12382 None for *flockfile()* and *funlockfile()*.

12383 The *ftrylockfile()* function shall return zero for success and non-zero to indicate that the lock
12384 cannot be acquired.

12385 ERRORS

12386 No errors are defined.

12387 EXAMPLES

12388 None.

12389 APPLICATION USAGE

12390 Applications using these functions may be subject to priority inversion, as discussed in the Base
12391 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

12392 RATIONALE

12393 The *flockfile()* and *funlockfile()* functions provide an orthogonal mutual-exclusion lock for each
12394 **FILE**. The *ftrylockfile()* function provides a non-blocking attempt to acquire a file lock,
12395 analogous to *pthread_mutex_trylock()*.

12396 These locks behave as if they are the same as those used internally by *stdio* for thread-safety.
12397 This both provides thread-safety of these functions without requiring a second level of internal
12398 locking and allows functions in *stdio* to be implemented in terms of other *stdio* functions.

12399 Application writers and implementors should be aware that there are potential deadlock
12400 problems on FILE objects. For example, the line-buffered flushing semantics of *stdio* (requested
12401 via {_IOLBF}) require that certain input operations sometimes cause the buffered contents of
12402 implementation-defined line-buffered output streams to be flushed. If two threads each hold the
12403 lock on the other's FILE, deadlock ensues. This type of deadlock can be avoided by acquiring
12404 FILE locks in a consistent order. In particular, the line-buffered output stream deadlock can
12405 typically be avoided by acquiring locks on input streams before locks on output streams if a
12406 thread would be acquiring both.

12407 In summary, threads sharing *stdio* streams with other threads can use *flockfile()* and *funlockfile()*
12408 to cause sequences of I/O performed by a single thread to be kept bundled. The only case where
12409 the use of *flockfile()* and *funlockfile()* is required is to provide a scope protecting uses of the
12410 *_unlocked() functions/macros. This moves the cost/performance tradeoff to the optimal point.

12411 FUTURE DIRECTIONS

12412 None.

12413 SEE ALSO

12414 *getc_unlocked()*, *putc_unlocked()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

12415 CHANGE HISTORY

12416 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

12417 Issue 6

12418 These functions are marked as part of the Thread-Safe Functions option.

12419 NAME

12420 floor, floorf, floorl — floor function

12421 SYNOPSIS

```
12422     #include <math.h>
12423
12424     double floor(double x);
12425     float floorf(float x);
12426     long double floorl(long double x);
```

12426 DESCRIPTION

12427 CX 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.

12430 These functions shall compute the largest integral value not greater than *x*.

12431 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.

12435 RETURN VALUE

12436 Upon successful completion, these functions shall return the largest integral value not greater than *x*, expressed as a **double**, **float**, or **long double**, as appropriate for the return type of the function.

12439 MX If *x* is NaN, a NaN shall be returned.

12440 If *x* is ±0 or ±Inf, *x* shall be returned.

12441 XSI If the correct value would cause overflow, a range error shall occur and *floor()*, *floorf()*, and *floorl()* shall return the value of the macro **-HUGE_VAL**, **-HUGE_VALF**, and **-HUGE_VALL**, respectively.

12444 ERRORS

12445 These functions shall fail if:

12446 XSI Range Error The result would cause an overflow.

12447 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.

12451 EXAMPLES

12452 None.

12453 APPLICATION USAGE

12454 The integral value returned by these functions might not be expressible as an **int** or **long**. The return value should be tested before assigning it to an integer type to avoid the undefined results of an integer overflow.

12455 The *floor()* function can only overflow when the floating-point representation has **DBL_MANT_DIG > DBL_MAX_EXP**.

12456 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

12461 RATIONALE

12462 None.

12463 FUTURE DIRECTIONS

12464 None.

12465 SEE ALSO

12466 *ceil()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
12467 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

12468 CHANGE HISTORY

12469 First released in Issue 1. Derived from Issue 1 of the SVID.

12470 Issue 5

12471 The DESCRIPTION is updated to indicate how an application should check for an error. This
12472 text was previously published in the APPLICATION USAGE section.

12473 Issue 6

12474 The *floorf()* and *floorl()* functions are added for alignment with the ISO/IEC 9899: 1999 standard.

12475 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
12476 revised to align with the ISO/IEC 9899: 1999 standard.

12477 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are
12478 marked.

12479 NAME

12480 fma, fmaf, fmal — floating-point multiply-add

12481 SYNOPSIS

```
12482 #include <math.h>
12483 double fma(double x, double y, double z);
12484 float fmaf(float x, float y, float z);
12485 long double fmal(long double x, long double y, long double z);
```

12486 DESCRIPTION

12487 CX 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.

12490 These functions shall compute $(x * y) + z$, rounded as one ternary operation: they shall compute the value (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value of `FLT_ROUNDS`.

12493 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.

12497 RETURN VALUE

12498 Upon successful completion, these functions shall return $(x * y) + z$, rounded as one ternary operation.

12500 MX If x or y are NaN, a NaN shall be returned.

12501 If x multiplied by y is an exact infinity and z is also an infinity but with the opposite sign, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

12504 If one of x and y is infinite, the other is zero, and z is not a NaN, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

12506 If one of x and y is infinite, the other is zero, and z is a NaN, a NaN shall be returned and a domain error may occur.

12508 If $x * y$ is not $0 * \text{Inf}$ nor $\text{Inf} * 0$ and z is a NaN, a NaN shall be returned.

12509 ERRORS

12510 These functions shall fail if:

12511 MX Domain Error The value of $x * y + z$ is invalid, or the value $x * y$ is invalid and z is not a NaN.

12512 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.

12516 MX Range Error The result overflows.

12517 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.

12521	These functions may fail if:	
12522 MX	Domain Error	The value $x*y$ is invalid and z is a NaN.
12523		If the integer expression ($\text{math_errhandling} \& \text{MATH_ERRNO}$) is non-zero, then errno shall be set to [EDOM]. If the integer expression ($\text{math_errhandling} \& \text{MATH_ERREXCEPT}$) is non-zero, then the invalid floating-point exception shall be raised.
12524		
12525		
12526		
12527 MX	Range Error	The result underflows.
12528		If the integer expression ($\text{math_errhandling} \& \text{MATH_ERRNO}$) is non-zero, then errno shall be set to [ERANGE]. If the integer expression ($\text{math_errhandling} \& \text{MATH_ERREXCEPT}$) is non-zero, then the underflow floating-point exception shall be raised.
12529		
12530		
12531		

12532 EXAMPLES

12533 None.

12534 APPLICATION USAGE

12535 On error, the expressions ($\text{math_errhandling} \& \text{MATH_ERRNO}$) and ($\text{math_errhandling} \& \text{MATH_ERREXCEPT}$) are independent of each other, but at least one of them must be non-zero.

12537 RATIONALE

12538 In many cases, clever use of floating (*fused*) multiply-add leads to much improved code; but its
12539 unexpected use by the compiler can undermine carefully written code. The FP_CONTRACT
12540 macro can be used to disallow use of floating multiply-add; and the *fma()* function guarantees
12541 its use where desired. Many current machines provide hardware floating multiply-add
12542 instructions; software implementation can be used for others.

12543 FUTURE DIRECTIONS

12544 None.

12545 SEE ALSO

12546 *feclearexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18,
12547 Treatment of Error Conditions for Mathematical Functions, <**math.h**>

12548 CHANGE HISTORY

12549 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

12550 NAME

12551 `fmax, fmaxf, fmaxl` — determine maximum numeric value of two floating-point numbers

12552 SYNOPSIS

```
12553        #include <math.h>
12554
12555        double fmax(double x, double y);
12556        float fmaxf(float x, float y);
12557        long double fmaxl(long double x, long double y);
```

12557 DESCRIPTION

12558 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12559 conflict between the requirements described here and the ISO C standard is unintentional. This
12560 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12561 These functions shall determine the maximum numeric value of their arguments. NaN
12562 arguments shall be treated as missing data: if one argument is a NaN and the other numeric,
12563 then these functions shall choose the numeric value.

12564 RETURN VALUE

12565 Upon successful completion, these functions shall return the maximum numeric value of their
12566 arguments.

12567 If just one argument is a NaN, the other argument shall be returned.

12568 MX If `x` and `y` are NaN, a NaN shall be returned.

12569 ERRORS

12570 No errors are defined.

12571 EXAMPLES

12572 None.

12573 APPLICATION USAGE

12574 None.

12575 RATIONALE

12576 None.

12577 FUTURE DIRECTIONS

12578 None.

12579 SEE ALSO

12580 `fdim()`, `fmin()`, the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

12581 CHANGE HISTORY

12582 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

12583 NAME

12584 `fmin, fminf, fminl` — determine minimum numeric value of two floating-point numbers

12585 SYNOPSIS

```
12586        #include <math.h>
12587
12588        double fmin(double x, double y);
12589        float fminf(float x, float y);
12590        long double fminl(long double x, long double y);
```

12590 DESCRIPTION

12591 CX The functionality described on this reference page is aligned with the ISO C standard. Any
12592 conflict between the requirements described here and the ISO C standard is unintentional. This
12593 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12594 These functions shall determine the minimum numeric value of their arguments. NaN
12595 arguments shall be treated as missing data: if one argument is a NaN and the other numeric,
12596 then these functions shall choose the numeric value.

12597 RETURN VALUE

12598 Upon successful completion, these functions shall return the minimum numeric value of their
12599 arguments.

12600 If just one argument is a NaN, the other argument shall be returned.

12601 MX If `x` and `y` are NaN, a NaN shall be returned.

12602 ERRORS

12603 No errors are defined.

12604 EXAMPLES

12605 None.

12606 APPLICATION USAGE

12607 None.

12608 RATIONALE

12609 None.

12610 FUTURE DIRECTIONS

12611 None.

12612 SEE ALSO

12613 `fdim()`, `fmax()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<math.h>`

12614 CHANGE HISTORY

12615 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

12616 NAME

12617 fmod, fmodf, fmodl — floating-point remainder value function

12618 SYNOPSIS

```
12619 #include <math.h>
12620 double fmod(double x, double y);
12621 float fmodf(float x, float y);
12622 long double fmodl(long double x, long double y);
```

12623 DESCRIPTION

12624 CX 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.

12627 These functions shall return the floating-point remainder of the division of x by y .

12628 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.

12632 RETURN VALUE

12633 These functions shall return the value $x - i*y$, for some integer i such that, if y is non-zero, the result has the same sign as x and magnitude less than the magnitude of y .

12635 If the correct value would cause underflow, and is not representable, a range error may occur, and either 0.0 (if supported), or an implementation-defined value shall be returned.

12637 MX If x or y is NaN, a NaN shall be returned.

12638 If y is zero, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

12640 If x is infinite, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

12642 If x is ± 0 and y is not zero, ± 0 shall be returned.

12643 If x is not infinite and y is $\pm\infty$, x shall be returned.

12644 If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

12646 ERRORS

12647 These functions shall fail if:

12648 MX Domain Error The x argument is infinite or y is zero.

12649 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.

12653 These functions may fail if:

12654 Range Error The result underflows.

12655 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 underflow floating-point exception shall be raised.

12659 EXAMPLES

12660 None.

12661 APPLICATION USAGE

12662 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
12663 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

12664 RATIONALE

12665 None.

12666 FUTURE DIRECTIONS

12667 None.

12668 SEE ALSO

12669 *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
12670 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

12671 CHANGE HISTORY

12672 First released in Issue 1. Derived from Issue 1 of the SVID.

12673 Issue 5

12674 The DESCRIPTION is updated to indicate how an application should check for an error. This
12675 text was previously published in the APPLICATION USAGE section.

12676 Issue 6

12677 The behavior for when the y argument is zero is now defined.

12678 The *fmodf()* and *fmodl()* functions are added for alignment with the ISO/IEC 9899:1999
12679 standard.

12680 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
12681 revised to align with the ISO/IEC 9899:1999 standard.

12682 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
12683 marked.

12684 NAME

12685 fmtmsg — display a message in the specified format on standard error and/or a system console

12686 SYNOPSIS

12687 XSI #include <fmtmsg.h>

```
12688        int fmtmsg(long classification, const char *label, int severity,  
12689                    const char *text, const char *action, const char *tag);  
12690
```

12691 DESCRIPTION

12692 The *fmtmsg()* function shall display messages in a specified format instead of the traditional
12693 *printf()* function.

12694 Based on a message's classification component, *fmtmsg()* shall write a formatted message either
12695 to standard error, to the console, or to both.

12696 A formatted message consists of up to five components as defined below. The component
12697 *classification* is not part of a message displayed to the user, but defines the source of the message
12698 and directs the display of the formatted message.

12699 *classification* Contains the sum of identifying values constructed from the constants defined
12700 below. Any one identifier from a subclass may be used in combination with a
12701 single identifier from a different subclass. Two or more identifiers from the
12702 same subclass should not be used together, with the exception of identifiers
12703 from the display subclass. (Both display subclass identifiers may be used so
12704 that messages can be displayed to both standard error and the system
12705 console.)

12706 **Major Classifications**

12707 Identifies the source of the condition. Identifiers are: MM_HARD
12708 (hardware), MM_SOFT (software), and MM_FIRM (firmware).

12709 **Message Source Subclassifications**

12710 Identifies the type of software in which the problem is detected.
12711 Identifiers are: MM_APPL (application), MM_UTIL (utility), and
12712 MM_OPSYS (operating system).

12713 **Display Subclassifications**

12714 Indicates where the message is to be displayed. Identifiers are:
12715 MM_PRINT to display the message on the standard error stream,
12716 MM_CONSOLE to display the message on the system console. One or
12717 both identifiers may be used.

12718 **Status Subclassifications**

12719 Indicates whether the application can recover from the condition.
12720 Identifiers are: MM_RECOVER (recoverable) and MM_NRECOV (non-
12721 recoverable).

12722 An additional identifier, MM_NULLMC, indicates that no classification
12723 component is supplied for the message.

12724 *label* Identifies the source of the message. The format is two fields separated by a
12725 colon. The first field is up to 10 bytes, the second is up to 14 bytes.

12726 *severity* Indicates the seriousness of the condition. Identifiers for the levels of *severity*
12727 are:

12728	MM_HALT	Indicates that the application has encountered a severe fault and is halting. Produces the string "HALT".
12729		
12730	MM_ERROR	Indicates that the application has detected a fault. Produces the string "ERROR".
12731		
12732	MM_WARNING	Indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the string "WARNING".
12733		
12734		
12735	MM_INFO	Provides information about a condition that is not in error. Produces the string "INFO".
12736		
12737	MM_NOSEV	Indicates that no severity level is supplied for the message.
12738	<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.
12739		
12740		
12741	<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.
12742		
12743		
12744	<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".
12745		
12746		
12747	The <i>MSGVERB</i> environment variable (for message verbosity) shall determine for <i>fmtmsg()</i> which message components it is to select when writing messages to standard error. The value of <i>MSGVERB</i> shall be a colon-separated list of optional keywords. Valid keywords are: <i>label</i> , <i>severity</i> , <i>text</i> , <i>action</i> , and <i>tag</i> . If <i>MSGVERB</i> contains a keyword for a component and the component's value is not the component's null value, <i>fmtmsg()</i> shall include that component in the message when writing the message to standard error. If <i>MSGVERB</i> does not include a keyword for a message component, that component shall not be included in the display of the message. The keywords may appear in any order. If <i>MSGVERB</i> is not defined, if its value is the null string, if its value is not of the correct format, or if it contains keywords other than the valid ones listed above, <i>fmtmsg()</i> shall select all components.	
12748		
12749		
12750		
12751		
12752		
12753		
12754		
12755		
12756		
12757	MSGVERB	shall determine which components are selected for display to standard error. All message components shall be included in console messages.
12758		
12759	RETURN VALUE	
12760	The <i>fmtmsg()</i> function shall return one of the following values:	
12761	MM_OK	The function succeeded.
12762	MM_NOTOK	The function failed completely.
12763	MM_NOMSG	The function was unable to generate a message on standard error, but otherwise succeeded.
12764		
12765	MM_NOCON	The function was unable to generate a console message, but otherwise succeeded.
12766		
12767	ERRORS	
12768	None.	

12769 EXAMPLES

- 12770 1. The following example of *fmtmsg()*:

12771 *fmtmsg(MM_PRINT, "XSI:cat", MM_ERROR, "illegal option",*
12772 *"refer to cat in user's reference manual", "XSI:cat:001")*

12773 produces a complete message in the specified message format:

12774 XSI:cat: ERROR: illegal option
12775 TO FIX: refer to cat in user's reference manual XSI:cat:001

- 12776 2. When the environment variable *MSGVERB* is set as follows:

12777 *MSGVERB=severity:text:action*

12778 and Example 1 is used, *fmtmsg()* produces:

12779 *ERROR: illegal option*
12780 *TO FIX: refer to cat in user's reference manual*

12781 APPLICATION USAGE

12782 One or more message components may be systematically omitted from messages generated by
12783 an application by using the null value of the argument for that component.

12784 RATIONALE

12785 None.

12786 FUTURE DIRECTIONS

12787 None.

12788 SEE ALSO

12789 *printf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fmtmsg.h>

12790 CHANGE HISTORY

12791 First released in Issue 4, Version 2.

12792 Issue 5

12793 Moved from X/OPEN UNIX extension to BASE.

12794 NAME

12795 fnmatch — match a filename or a pathname

12796 SYNOPSIS

12797 #include <fnmatch.h>
12798 int fnmatch(const char *pattern, const char *string, int flags);

12799 DESCRIPTION

12800 The *fnmatch()* function shall match patterns as described in the Shell and Utilities volume of
12801 IEEE Std 1003.1-2001, Section 2.13.1, Patterns Matching a Single Character, and Section 2.13.2,
12802 Patterns Matching Multiple Characters. It checks the string specified by the *string* argument to
12803 see if it matches the pattern specified by the *pattern* argument.

12804 The *flags* argument shall modify the interpretation of *pattern* and *string*. It is the bitwise-inclusive
12805 OR of zero or more of the flags defined in <fnmatch.h>. If the FNM_PATHNAME flag is set in
12806 *flags*, then a slash character (' / ') in *string* shall be explicitly matched by a slash in *pattern*; it shall
12807 not be matched by either the asterisk or question-mark special characters, nor by a bracket
12808 expression. If the FNM_PATHNAME flag is not set, the slash character shall be treated as an
12809 ordinary character.

12810 If FNM_NOESCAPE is not set in *flags*, a backslash character (' \ ') in *pattern* followed by any
12811 other character shall match that second character in *string*. In particular, "\\" shall match a
12812 backslash in *string*. If FNM_NOESCAPE is set, a backslash character shall be treated as an
12813 ordinary character.

12814 If FNM_PERIOD is set in *flags*, then a leading period (' . ') in *string* shall match a period in
12815 *pattern*; as described by rule 2 in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section
12816 2.13.3, Patterns Used for Filename Expansion where the location of "leading" is indicated by the
12817 value of FNM_PATHNAME:

- 12818 • If FNM_PATHNAME is set, a period is "leading" if it is the first character in *string* or if it
12819 immediately follows a slash.
- 12820 • If FNM_PATHNAME is not set, a period is "leading" only if it is the first character of *string*.

12821 If FNM_PERIOD is not set, then no special restrictions are placed on matching a period.

12822 RETURN VALUE

12823 If *string* matches the pattern specified by *pattern*, then *fnmatch()* shall return 0. If there is no
12824 match, *fnmatch()* shall return FNM_NOMATCH, which is defined in <fnmatch.h>. If an error
12825 occurs, *fnmatch()* shall return another non-zero value.

12826 ERRORS

12827 No errors are defined.

12828 EXAMPLES

12829 None.

12830 APPLICATION USAGE

12831 The *fnmatch()* function has two major uses. It could be used by an application or utility that
12832 needs to read a directory and apply a pattern against each entry. The *find* utility is an example of
12833 this. It can also be used by the *pax* utility to process its *pattern* operands, or by applications that
12834 need to match strings in a similar manner.

12835 The name *fnmatch()* is intended to imply *filename* match, rather than *pathname* match. The default
12836 action of this function is to match filenames, rather than pathnames, since it gives no special
12837 significance to the slash character. With the FNM_PATHNAME flag, *fnmatch()* does match
12838 pathnames, but without tilde expansion, parameter expansion, or special treatment for a period

12839 at the beginning of a filename.

12840 **RATIONALE**

12841 This function replaced the REG_FILENAME flag of *regcomp()* in early proposals of this volume
12842 of IEEE Std 1003.1-2001. It provides virtually the same functionality as the *regcomp()* and
12843 *regexec()* functions using the REG_FILENAME and REG_FSLASH flags (the REG_FSLASH flag
12844 was proposed for *regcomp()*, and would have had the opposite effect from FNM_PATHNAME),
12845 but with a simpler function and less system overhead.

12846 **FUTURE DIRECTIONS**

12847 None.

12848 **SEE ALSO**

12849 *glob()*, *wordexp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fnmatch.h>, the Shell
12850 and Utilities volume of IEEE Std 1003.1-2001

12851 **CHANGE HISTORY**

12852 First released in Issue 4. Derived from the ISO POSIX-2 standard.

12853 **Issue 5**

12854 Moved from POSIX2 C-language Binding to BASE.

12855 NAME

12856 *fopen* — open a stream

12857 SYNOPSIS

```
12858        #include <stdio.h>
12859        FILE *fopen(const char *restrict filename, const char *restrict mode);
```

12860 DESCRIPTION

12861 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 12862 conflict between the requirements described here and the ISO C standard is unintentional. This
 12863 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

12864 The *fopen()* function shall open the file whose pathname is the string pointed to by *filename*, and
 12865 associates a stream with it.

12866 The *mode* argument points to a string. If the string is one of the following, the file shall be opened
 12867 in the indicated mode. Otherwise, the behavior is undefined.

12868 *r* or *rb* Open file for reading.

12869 *w* or *wb* Truncate to zero length or create file for writing.

12870 *a* or *ab* Append; open or create file for writing at end-of-file.

12871 *r+* or *rb+* or *r+b* Open file for update (reading and writing).

12872 *w+* or *wb+* or *w+b* Truncate to zero length or create file for update.

12873 *a+* or *ab+* or *a+b* Append; open or create file for update, writing at end-of-file.

12874 CX The character '*b*' shall have no effect, but is allowed for ISO C standard conformance. Opening
 12875 a file with read mode (*r* as the first character in the *mode* argument) shall fail if the file does not
 12876 exist or cannot be read.

12877 Opening a file with append mode (*a* as the first character in the *mode* argument) shall cause all
 12878 subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening
 12879 calls to *fseek()*.

12880 When a file is opened with update mode ('+' as the second or third character in the *mode*
 12881 argument), both input and output may be performed on the associated stream. However, the
 12882 application shall ensure that output is not directly followed by input without an intervening call
 12883 to *fflush()* or to a file positioning function (*fseek()*, *fsetpos()*, or *rewind()*), and input is not directly
 12884 followed by output without an intervening call to a file positioning function, unless the input
 12885 operation encounters end-of-file.

12886 When opened, a stream is fully buffered if and only if it can be determined not to refer to an
 12887 interactive device. The error and end-of-file indicators for the stream shall be cleared.

12888 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
 12889 successful completion, the *fopen()* function shall mark for update the *st_atime*, *st_ctime*, and
 12890 *st_mtime* fields of the file and the *st_ctime* and *st_mtime* fields of the parent directory.

12891 If *mode* is *w*, *wb*, *w+*, *wb+*, or *w+b*, and the file did previously exist, upon successful completion,
 12892 *fopen()* shall mark for update the *st_ctime* and *st_mtime* fields of the file. The *fopen()* function
 12893 shall allocate a file descriptor as *open()* does.

12894 XSI After a successful call to the *fopen()* function, the orientation of the stream shall be cleared, the
 12895 encoding rule shall be cleared, and the associated **mbstate_t** object shall be set to describe an
 12896 initial conversion state.

12897 CX The largest value that can be represented correctly in an object of type `off_t` shall be established
 12898 as the offset maximum in the open file description.

12899 RETURN VALUE

12900 Upon successful completion, `fopen()` shall return a pointer to the object controlling the stream.
 12901 CX Otherwise, a null pointer shall be returned, and `errno` shall be set to indicate the error.

12902 ERRORS

12903 The `fopen()` function shall fail if:

12904 CX [EACCES] Search permission is denied on a component of the path prefix, or the file
 12905 exists and the permissions specified by `mode` are denied, or the file does not
 12906 exist and write permission is denied for the parent directory of the file to be
 12907 created.

12908 CX [EINTR] A signal was caught during `fopen()`.

12909 CX [EISDIR] The named file is a directory and `mode` requires write access.

12910 CX [ELOOP] A loop exists in symbolic links encountered during resolution of the `path`
 12911 argument.

12912 CX [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

12913 CX [ENAMETOOLONG]

12914 The length of the `filename` argument exceeds {PATH_MAX} or a pathname
 12915 component is longer than {NAME_MAX}.

12916 CX [ENFILE] The maximum allowable number of files is currently open in the system.

12917 CX [ENOENT] A component of `filename` does not name an existing file or `filename` is an empty
 12918 string.

12919 CX [ENOSPC] The directory or file system that would contain the new file cannot be
 12920 expanded, the file does not exist, and the file was to be created.

12921 CX [ENOTDIR] A component of the path prefix is not a directory.

12922 CX [ENXIO] The named file is a character special or block special file, and the device
 12923 associated with this special file does not exist.

12924 CX [EOVERFLOW] The named file is a regular file and the size of the file cannot be represented
 12925 correctly in an object of type `off_t`.

12926 CX [EROFS] The named file resides on a read-only file system and `mode` requires write
 12927 access.

12928 The `fopen()` function may fail if:

12929 CX [EINVAL] The value of the `mode` argument is not valid.

12930 CX [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 12931 resolution of the `path` argument.

12932 CX [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

12933 CX [EMFILE] {STREAM_MAX} streams are currently open in the calling process.

12934 CX [ENAMETOOLONG]

12935 Pathname resolution of a symbolic link produced an intermediate result
 12936 whose length exceeds {PATH_MAX}.

12937 CX [ENOMEM]	Insufficient storage space is available.
12938 CX [ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and <i>mode</i> requires write access.
12939	

12940 EXAMPLES

12941 Opening a File

12942 The following example tries to open the file named **file** for reading. The *fopen()* function returns
12943 a file pointer that is used in subsequent *fgets()* and *fclose()* calls. If the program cannot open the
12944 file, it just ignores it.

```
12945 #include <stdio.h>
12946 ...
12947 FILE *fp;
12948 ...
12949 void rgrep(const char *file)
12950 {
12951 ...
12952     if ((fp = fopen(file, "r")) == NULL)
12953         return;
12954 ...
12955 }
```

12956 APPLICATION USAGE

12957 None.

12958 RATIONALE

12959 None.

12960 FUTURE DIRECTIONS

12961 None.

12962 SEE ALSO

12963 *fclose()*, *fdopen()*, *freopen()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

12964 CHANGE HISTORY

12965 First released in Issue 1. Derived from Issue 1 of the SVID.

12966 Issue 5

12967 Large File Summit extensions are added.

12968 Issue 6

12969 Extensions beyond the ISO C standard are marked.

12970 The following new requirements on POSIX implementations derive from alignment with the
12971 Single UNIX Specification:

- 12972 • In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file
12973 description. This change is to support large files.
- 12974 • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support
12975 large files.
- 12976 • The [ELOOP] mandatory error condition is added.
- 12977 • The [EINVAL], [EMFILE], [ENAMETOOLONG], [ENOMEM], and [ETXTBSY] optional error
12978 conditions are added.

12979 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

12980 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

12981 • The prototype for *fopen()* is updated.

12982 • The DESCRIPTION is updated to note that if the argument *mode* points to a string other than
12983 those listed, then the behavior is undefined.

12984 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
12985 [ELOOP] error condition is added.

12986 NAME

12987 fork — create a new process

12988 SYNOPSIS

12989 #include <unistd.h>
12990 pid_t fork(void);

12991 DESCRIPTION

12992 The *fork()* function shall create a new process. The new process (child process) shall be an exact
12993 copy of the calling process (parent process) except as detailed below:

- 12994 • The child process shall have a unique process ID.
- 12995 • The child process ID also shall not match any active process group ID.
- 12996 • The child process shall have a different parent process ID, which shall be the process ID of
12997 the calling process.
- 12998 • The child process shall have its own copy of the parent's file descriptors. Each of the child's
12999 file descriptors shall refer to the same open file description with the corresponding file
13000 descriptor of the parent.
- 13001 • The child process shall have its own copy of the parent's open directory streams. Each open
13002 directory stream in the child process may share directory stream positioning with the
13003 corresponding directory stream of the parent.

13004 XSI

- 13005 • The child process shall have its own copy of the parent's message catalog descriptors.
- 13006 • The child process' values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* shall be set to 0.
- 13007 • The time left until an alarm clock signal shall be reset to zero, and the alarm, if any, shall be
canceled; see *alarm()*.

13008 XSI

- 13009 • All *semadj* values shall be cleared.
- 13010 • File locks set by the parent process shall not be inherited by the child process.

13011 XSI

- 13012 • The set of signals pending for the child process shall be initialized to the empty set.

13012 SEM

- 13013 ML • Any semaphores that are open in the parent process shall also be open in the child process.

13014

- 13015 MF|SHM • The child process shall not inherit any address space memory locks established by the parent
process via calls to *mlockall()* or *mlock()*.

13016

- 13017 • Memory mappings created in the parent shall be retained in the child process.
13018 MAP_PRIVATE mappings inherited from the parent shall also be MAP_PRIVATE mappings
13019 in the child, and any modifications to the data in these mappings made by the parent prior to
13020 calling *fork()* shall be visible to the child. Any modifications to the data in MAP_PRIVATE
13021 mappings made by the parent after *fork()* returns shall be visible only to the parent.
Modifications to the data in MAP_PRIVATE mappings made by the child shall be visible only
to the child.

13022 PS

- 13023 • For the SCHED_FIFO and SCHED_RR scheduling policies, the child process shall inherit the
13024 policy and priority settings of the parent process during a *fork()* function. For other
scheduling policies, the policy and priority settings on *fork()* are implementation-defined.

13025 TMR

- 13026 MSG • Per-process timers created by the parent shall not be inherited by the child process.

13027

- 13028 • The child process shall have its own copy of the message queue descriptors of the parent.
Each of the message descriptors of the child shall refer to the same open message queue

13028	description as the corresponding message descriptor of the parent.	
13029 AIO	• No asynchronous input or asynchronous output operations shall be inherited by the child process.	
13030		
13031	• A process shall be created with a single thread. If a multi-threaded process calls <i>fork()</i> , the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, to avoid errors, the child process may only execute async-signal-safe operations until such time as one of the <i>exec</i> functions is called. Fork handlers may be established by means of the <i>pthread_atfork()</i> function in order to maintain application invariants across <i>fork()</i> calls.	
13032		
13033		
13034		
13035 THR		
13036		
13037	When the application calls <i>fork()</i> from a signal handler and any of the fork handlers registered by <i>pthread_atfork()</i> calls a function that is not asynch-signal-safe, the behavior is undefined.	1
13038		1
13039		1
13040 TRC TRI	• If the Trace option and the Trace Inherit option are both supported:	
13041	If the calling process was being traced in a trace stream that had its inheritance policy set to POSIX_TRACE_INHERITED, the child process shall be traced into that trace stream, and the child process shall inherit the parent's mapping of trace event names to trace event type identifiers. If the trace stream in which the calling process was being traced had its inheritance policy set to POSIX_TRACE_CLOSE_FOR_CHILD, the child process shall not be traced into that trace stream. The inheritance policy is set by a call to the <i>posix_trace_attr_setinherited()</i> function.	
13042		
13043		
13044		
13045		
13046		
13047		
13048 TRC	• If the Trace option is supported, but the Trace Inherit option is not supported:	
13049	The child process shall not be traced into any of the trace streams of its parent process.	
13050 TRC	• If the Trace option is supported, the child process of a trace controller process shall not control the trace streams controlled by its parent process.	
13051		
13052 CPT	• The initial value of the CPU-time clock of the child process shall be set to zero.	
13053 TCT	• The initial value of the CPU-time clock of the single thread of the child process shall be set to zero.	
13054		
13055	All other process characteristics defined by IEEE Std 1003.1-2001 shall be the same in the parent and child processes. The inheritance of process characteristics not defined by IEEE Std 1003.1-2001 is unspecified by IEEE Std 1003.1-2001.	
13056		
13057		
13058	After <i>fork()</i> , both the parent and the child processes shall be capable of executing independently before either one terminates.	
13059		
13060	RETURN VALUE	
13061	Upon successful completion, <i>fork()</i> shall return 0 to the child process and shall return the process ID of the child process to the parent process. Both processes shall continue to execute from the <i>fork()</i> function. Otherwise, -1 shall be returned to the parent process, no child process shall be created, and <i>errno</i> shall be set to indicate the error.	
13062		
13063		
13064		
13065	ERRORS	
13066	The <i>fork()</i> function shall fail if:	
13067	[EAGAIN]	The system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution system-wide or by a single user {CHILD_MAX} would be exceeded.
13068		
13069		

13070 The *fork()* function may fail if:
13071 [ENOMEM] Insufficient storage space is available.

13072 EXAMPLES

13073 None.

13074 APPLICATION USAGE

13075 None.

13076 RATIONALE

13077 Many historical implementations have timing windows where a signal sent to a process group
13078 (for example, an interactive SIGINT) just prior to or during execution of *fork()* is delivered to the
13079 parent following the *fork()* but not to the child because the *fork()* code clears the child's set of
13080 pending signals. This volume of IEEE Std 1003.1-2001 does not require, or even permit, this
13081 behavior. However, it is pragmatic to expect that problems of this nature may continue to exist
13082 in implementations that appear to conform to this volume of IEEE Std 1003.1-2001 and pass
13083 available verification suites. This behavior is only a consequence of the implementation failing to
13084 make the interval between signal generation and delivery totally invisible. From the
13085 application's perspective, a *fork()* call should appear atomic. A signal that is generated prior to
13086 the *fork()* should be delivered prior to the *fork()*. A signal sent to the process group after the
13087 *fork()* should be delivered to both parent and child. The implementation may actually initialize
13088 internal data structures corresponding to the child's set of pending signals to include signals
13089 sent to the process group during the *fork()*. Since the *fork()* call can be considered as atomic
13090 from the application's perspective, the set would be initialized as empty and such signals would
13091 have arrived after the *fork()*; see also <signal.h>.

13092 One approach that has been suggested to address the problem of signal inheritance across *fork()*
13093 is to add an [EINTR] error, which would be returned when a signal is detected during the call.
13094 While this is preferable to losing signals, it was not considered an optimal solution. Although it
13095 is not recommended for this purpose, such an error would be an allowable extension for an
13096 implementation.

13097 The [ENOMEM] error value is reserved for those implementations that detect and distinguish
13098 such a condition. This condition occurs when an implementation detects that there is not enough
13099 memory to create the process. This is intended to be returned when [EAGAIN] is inappropriate
13100 because there can never be enough memory (either primary or secondary storage) to perform the
13101 operation. Since *fork()* duplicates an existing process, this must be a condition where there is
13102 sufficient memory for one such process, but not for two. Many historical implementations
13103 actually return [ENOMEM] due to temporary lack of memory, a case that is not generally
13104 distinct from [EAGAIN] from the perspective of a conforming application.

13105 Part of the reason for including the optional error [ENOMEM] is because the SVID specifies it
13106 and it should be reserved for the error condition specified there. The condition is not applicable
13107 on many implementations.

13108 IEEE Std 1003.1-1988 neglected to require concurrent execution of the parent and child of *fork()*.
13109 A system that single-threads processes was clearly not intended and is considered an
13110 unacceptable "toy implementation" of this volume of IEEE Std 1003.1-2001. The only objection
13111 anticipated to the phrase "executing independently" is testability, but this assertion should be
13112 testable. Such tests require that both the parent and child can block on a detectable action of the
13113 other, such as a write to a pipe or a signal. An interactive exchange of such actions should be
13114 possible for the system to conform to the intent of this volume of IEEE Std 1003.1-2001.

13115 The [EAGAIN] error exists to warn applications that such a condition might occur. Whether it
13116 occurs or not is not in any practical sense under the control of the application because the
13117 condition is usually a consequence of the user's use of the system, not of the application's code.

13118 Thus, no application can or should rely upon its occurrence under any circumstances, nor
13119 should the exact semantics of what concept of “user” is used be of concern to the application
13120 writer. Validation writers should be cognizant of this limitation.

13121 There are two reasons why POSIX programmers call *fork()*. One reason is to create a new thread
13122 of control within the same program (which was originally only possible in POSIX by creating a
13123 new process); the other is to create a new process running a different program. In the latter case,
13124 the call to *fork()* is soon followed by a call to one of the *exec* functions.

13125 The general problem with making *fork()* work in a multi-threaded world is what to do with all
13126 of the threads. There are two alternatives. One is to copy all of the threads into the new process.
13127 This causes the programmer or implementation to deal with threads that are suspended on
13128 system calls or that might be about to execute system calls that should not be executed in the
13129 new process. The other alternative is to copy only the thread that calls *fork()*. This creates the
13130 difficulty that the state of process-local resources is usually held in process memory. If a thread
13131 that is not calling *fork()* holds a resource, that resource is never released in the child process
13132 because the thread whose job it is to release the resource does not exist in the child process.

13133 When a programmer is writing a multi-threaded program, the first described use of *fork()*,
13134 creating new threads in the same program, is provided by the *pthread_create()* function. The
13135 *fork()* function is thus used only to run new programs, and the effects of calling functions that
13136 require certain resources between the call to *fork()* and the call to an *exec* function are undefined.

13137 The addition of the *forkall()* function to the standard was considered and rejected. The *forkall()*
13138 function lets all the threads in the parent be duplicated in the child. This essentially duplicates
13139 the state of the parent in the child. This allows threads in the child to continue processing and
13140 allows locks and the state to be preserved without explicit *pthread_atfork()* code. The calling
13141 process has to ensure that the threads processing state that is shared between the parent and
13142 child (that is, file descriptors or MAP_SHARED memory) behaves properly after *forkall()*. For
13143 example, if a thread is reading a file descriptor in the parent when *forkall()* is called, then two
13144 threads (one in the parent and one in the child) are reading the file descriptor after the *forkall()*.
13145 If this is not desired behavior, the parent process has to synchronize with such threads before
13146 calling *forkall()*.

13147 While the *fork()* function is async-signal-safe, there is no way for an implementation to 1
13148 determine whether the fork handlers established by *pthread_atfork()* are async-signal-safe. The 1
13149 fork handlers may attempt to execute portions of the implementation that are not async-signal- 1
13150 safe, such as those that are protected by mutexes, leading to a deadlock condition. It is therefore 1
13151 undefined for the fork handlers to execute functions that are not async-signal-safe when *fork()* is 1
13152 called from a signal handler. 1

13153 When *forkall()* is called, threads, other than the calling thread, that are in functions that can 1
13154 return with an [EINTR] error may have those functions return [EINTR] if the implementation 1
13155 cannot ensure that the function behaves correctly in the parent and child. In particular, 1
13156 *pthread_cond_wait()* and *pthread_cond_timedwait()* need to return in order to ensure that the 1
13157 condition has not changed. These functions can be awakened by a spurious condition wakeup 1
13158 rather than returning [EINTR].

13159 FUTURE DIRECTIONS

13160 None.

13161 SEE ALSO

13162 *alarm()*, *exec*, *fcntl()*, *posix_trace_attr_getinherited()*, *posix_trace_trid_eventid_open()*, 1
13163 *pthread_atfork()*, *semop()*, *signal()*, *times()*, the Base Definitions volume of IEEE Std 1003.1-2001, 1
13164 <sys/types.h>, <unistd.h>

13165 **CHANGE HISTORY**

13166 First released in Issue 1. Derived from Issue 1 of the SVID.

13167 **Issue 5**13168 The DESCRIPTION is changed for alignment with the POSIX Realtime Extension and the POSIX
13169 Threads Extension.13170 **Issue 6**13171 The following new requirements on POSIX implementations derive from alignment with the
13172 Single UNIX Specification:

- 13173 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
-
- 13174 required for conforming implementations of previous POSIX specifications, it was not
-
- 13175 required for UNIX applications.

13176 The following changes were made to align with the IEEE P1003.1a draft standard:

- 13177 • The effect of
- fork()*
- on a pending alarm call in the child process is clarified.

13178 The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.

13179 The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

13180 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/17 is applied, adding text to the 1
13181 DESCRIPTION and RATIONALE relating to fork handlers registered by the *pthread_atfork()* 1
13182 function and async-signal safety.

13183 NAME

13184 fpathconf, pathconf — get configurable pathname variables

13185 SYNOPSIS

```
13186 #include <unistd.h>
13187 long fpathconf(int fildes, int name);
13188 long pathconf(const char *path, int name);
```

13189 DESCRIPTION

The *fpathconf()* and *pathconf()* functions shall determine the current value of a configurable limit or option (*variable*) that is associated with a file or directory.

13192 For *pathconf()*, the *path* argument points to the pathname of a file or directory.

13193 For *fpathconf()*, the *fildes* argument is an open file descriptor.

13194 The *name* argument represents the variable to be queried relative to that file or directory. Implementations shall support all of the variables listed in the following table and may support others. The variables in the following table come from *<limits.h>* or *<unistd.h>* and the symbolic constants, defined in *<unistd.h>*, are the corresponding values used for *name*.
1

13198

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
{POSIX2_SYMLINKS}	_PC_2_SYMLINKS	4
{POSIX_ALLOC_SIZE_MIN}	_PC_ALLOC_SIZE_MIN	10
{POSIX_REC_INCR_XFER_SIZE}	_PC_REC_INCR_XFER_SIZE	10
{POSIX_REC_MAX_XFER_SIZE}	_PC_REC_MAX_XFER_SIZE	10
{POSIX_REC_MIN_XFER_SIZE}	_PC_REC_MIN_XFER_SIZE	10
{POSIX_REC_XFER_ALIGN}	_PC_REC_XFER_ALIGN	10
{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

13220

Requirements

1. If *path* or *fildes* refers to a directory, the value returned shall apply to the directory itself.
2. If *path* or *fildes* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
3. If *path* or *fildes* refers to a directory, the value returned shall apply to filenames within the directory.
4. If *path* or *fildes* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.

5. If *path* or *fildes* refers to a directory, the value returned shall be the maximum length of a relative pathname when the specified directory is the working directory.
 6. If *path* refers to a FIFO, or *fildes* refers to a pipe or FIFO, the value returned shall apply to the referenced object. If *path* or *fildes* refers to a directory, the value returned shall apply to any FIFO that exists or can be created within the directory. If *path* or *fildes* refers to any other type of file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
 7. If *path* or *fildes* refers to a directory, the value returned shall apply to any files, other than directories, that exist or can be created within the directory.
 8. If *path* or *fildes* refers to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
 9. If *path* or *fildes* refers to a directory, the value returned shall be the maximum length of the string that a symbolic link in that directory can contain.
 10. If *path* or *fildes* does not refer to a regular file, it is unspecified whether an implementation supports an association of the variable name with the specified file. If an implementation supports such an association for other than a regular file, the value returned is unspecified.

13245 RETURN VALUE

13246 If *name* is an invalid value, both *pathconf()* and *fpathconf()* shall return -1 and set *errno* to indicate the error.
13247

If the variable corresponding to *name* has no limit for the *path* or file descriptor, both *pathconf()* and *fpathconf()* shall return -1 without changing *errno*. If the implementation needs to use *path* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *path*, or if the process did not have appropriate privileges to query the file specified by *path*, or *path* does not exist, *pathconf()* shall return -1 and set *errno* to indicate the error.

If the implementation needs to use *fildes* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *fildes*, or if *fildes* is an invalid file descriptor, *fpathconf()* shall return -1 and set *errno* to indicate the error.

Otherwise, *pathconf()* or *fpathconf()* shall return the current variable value for the file or directory without changing *errno*. The value returned shall not be more restrictive than the corresponding value available to the application when it was compiled with the implementation's <limits.h> or <unistd.h>.

13261 If the variable corresponding to *name* is dependent on an unsupported option, the results are 2
13262 unspecified. 2

13263 ERRORS

13264 The *pathconf()* function shall fail if:

13265 [EINVAL] The value of *name* is not valid.

13266 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.
13267

13268 The *pathconf()* function may fail if:

13269 [EACCES] Search permission is denied for a component of the path prefix.

13270 [EINVAL] The implementation does not support an association of the variable *name* with
13271 the specified file.

13272	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path</i> argument.
13274	[ENAMETOOLONG]	The length of the <i>path</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
13277	[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}.
13280	[ENOENT]	A component of <i>path</i> does not name an existing file or <i>path</i> is an empty string.
13281	[ENOTDIR]	A component of the path prefix is not a directory.
13282	The <i>fpathconf()</i> function shall fail if:	
13283	[EINVAL]	The value of <i>name</i> is not valid.
13284	The <i>fpathconf()</i> function may fail if:	
13285	[EBADF]	The <i>fd</i> argument is not a valid file descriptor.
13286	[EINVAL]	The implementation does not support an association of the variable <i>name</i> with the specified file.

13288 EXAMPLES

13289 None.

13290 APPLICATION USAGE

13291 Application writers should check whether an option, such as _POSIX_ADVISORY_INFO, is 2
13292 supported prior to obtaining and using values for related variables such as 2
13293 {POSIX_ALLOC_SIZE_MIN}. 2

13294 RATIONALE

13295 The *pathconf()* function was proposed immediately after the *sysconf()* function when it was
13296 realized that some configurable values may differ across file system, directory, or device
13297 boundaries.

13298 For example, {NAME_MAX} frequently changes between System V and BSD-based file systems;
13299 System V uses a maximum of 14, BSD 255. On an implementation that provides both types of file
13300 systems, an application would be forced to limit all pathname components to 14 bytes, as this
13301 would be the value specified in <limits.h> on such a system.

13302 Therefore, various useful values can be queried on any pathname or file descriptor, assuming
13303 that the appropriate permissions are in place.

13304 The value returned for the variable {PATH_MAX} indicates the longest relative pathname that
13305 could be given if the specified directory is the process' current working directory. A process may
13306 not always be able to generate a name that long and use it if a subdirectory in the pathname
13307 crosses into a more restrictive file system.

13308 The value returned for the variable _POSIX_CHOWN_RESTRICTED also applies to directories
13309 that do not have file systems mounted on them. The value may change when crossing a mount
13310 point, so applications that need to know should check for each directory. (An even easier check
13311 is to try the *chown()* function and look for an error in case it happens.)

13312 Unlike the values returned by *sysconf()*, the pathname-oriented variables are potentially more
13313 volatile and are not guaranteed to remain constant throughout the process' lifetime. For
13314 example, in between two calls to *pathconf()*, the file system in question may have been
13315 unmounted and remounted with different characteristics.

13316 Also note that most of the errors are optional. If one of the variables always has the same value
 13317 on an implementation, the implementation need not look at *path* or *fildes* to return that value and
 13318 is, therefore, not required to detect any of the errors except the meaning of [EINVAL] that
 13319 indicates that the value of *name* is not valid for that variable.

13320 If the value of any of the limits is unspecified (logically infinite), they will not be defined in
 13321 <limits.h> and the *pathconf()* and *fpathconf()* functions return -1 without changing *errno*. This
 13322 can be distinguished from the case of giving an unrecognized *name* argument because *errno* is set
 13323 to [EINVAL] in this case.

13324 Since -1 is a valid return value for the *pathconf()* and *fpathconf()* functions, applications should
 13325 set *errno* to zero before calling them and check *errno* only if the return value is -1.

13326 For the case of {SYMLINK_MAX}, since both *pathconf()* and *open()* follow symbolic links, there
 13327 is no way that *path* or *fildes* could refer to a symbolic link.

13328 It was the intention of IEEE Std 1003.1d-1999 that the following variables: 2

13329 {POSIX_ALLOC_SIZE_MIN}	2
13330 {POSIX_REC_INCR_XFER_SIZE}	2
13331 {POSIX_REC_MAX_XFER_SIZE}	2
13332 {POSIX_REC_MIN_XFER_SIZE}	2
13333 {POSIX_REC_XFER_ALIGN}	2

13334 only applied to regular files, but Note 10 also permits implementation of the advisory semantics 2
 13335 on other file types unique to an implementation (for example, a character special device). 2

13336 FUTURE DIRECTIONS

13337 None.

13338 SEE ALSO

13339 *confstr()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <unistd.h>,
 13340 the Shell and Utilities volume of IEEE Std 1003.1-2001, *getconf* 2

13341 CHANGE HISTORY

13342 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

13343 Issue 5

13344 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

13345 Large File Summit extensions are added.

13346 Issue 6

13347 The following new requirements on POSIX implementations derive from alignment with the
 13348 Single UNIX Specification:

- 13349 • The DESCRIPTION is updated to include {FILESIZEBITS}.
- 13350 • The [ELOOP] mandatory error condition is added.
- 13351 • A second [ENAMETOOLONG] is added as an optional error condition.

13352 The following changes were made to align with the IEEE P1003.1a draft standard:

- 13353 • The _PC_SYMLINK_MAX entry is added to the table in the DESCRIPTION.

13354 The following *pathconf()* variables and their associated names are added for alignment with
 13355 IEEE Std 1003.1d-1999:

13356	{POSIX_ALLOC_SIZE_MIN}	
13357	{POSIX_REC_INCR_XFER_SIZE}	
13358	{POSIX_REC_MAX_XFER_SIZE}	
13359	{POSIX_REC_MIN_XFER_SIZE}	
13360	{POSIX_REC_XFER_ALIGN}	
13361	IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/18 is applied, changing the fourth	1
13362	paragraph of the DESCRIPTION and removing shading and margin markers from the table. This	1
13363	change is needed since implementations are required to support all of these symbols.	1
13364	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/34 is applied, adding the table entry for	2
13365	POSIX2_SYMLINKS in the DESCRIPTION.	2
13366	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/35 is applied, updating the	2
13367	DESCRIPTION and RATIONALE sections to clarify behavior for the	2
13368	{POSIX_ALLOC_SIZE_MIN}, {POSIX_REC_INCR_XFER_SIZE},	2
13369	{POSIX_REC_MAX_XFER_SIZE}, {POSIX_REC_MIN_XFER_SIZE}, and	2
13370	{POSIX_REC_XFER_ALIGN} variables.	2
13371	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/36 is applied, updating the RETURN	2
13372	VALUE and APPLICATION USAGE sections to state that the results are unspecified if a variable	2
13373	is dependent on an unsupported option, and advising application writers to check for supported	2
13374	options prior to obtaining and using such values.	2

13375 NAME

13376 *fpclassify* — classify real floating type

13377 SYNOPSIS

```
13378        #include <math.h>  
13379        int fpclassify(real-floating x);
```

13380 DESCRIPTION

13381 CX The functionality described on this reference page is aligned with the ISO C standard. Any
13382 conflict between the requirements described here and the ISO C standard is unintentional. This
13383 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

13384 The *fpclassify()* macro shall classify its argument value as NaN, infinite, normal, subnormal,
13385 zero, or into another implementation-defined category. First, an argument represented in a
13386 format wider than its semantic type is converted to its semantic type. Then classification is based
13387 on the type of the argument.

13388 RETURN VALUE

13389 The *fpclassify()* macro shall return the value of the number classification macro appropriate to
13390 the value of its argument.

13391 ERRORS

13392 No errors are defined.

13393 EXAMPLES

13394 None.

13395 APPLICATION USAGE

13396 None.

13397 RATIONALE

13398 None.

13399 FUTURE DIRECTIONS

13400 None.

13401 SEE ALSO

13402 *isfinite()*, *isinf()*, *isnan()*, *isnormal()*, *signbit()*, the Base Definitions volume of
13403 IEEE Std 1003.1-2001, <math.h>

13404 CHANGE HISTORY

13405 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

13406 NAME

13407 fprintf, printf, snprintf, sprintf — print formatted output

13408 SYNOPSIS

```
13409     #include <stdio.h>
13410
13411     int fprintf(FILE *restrict stream, const char *restrict format, ...);
13412     int printf(const char *restrict format, ...);
13413     int snprintf(char *restrict s, size_t n,
13414         const char *restrict format, ...);
13415     int sprintf(char *restrict s, const char *restrict format, ...);
```

13415 DESCRIPTION

13416 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 13417 conflict between the requirements described here and the ISO C standard is unintentional. This
 13418 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

13419 The *fprintf()* function shall place output on the named output *stream*. The *printf()* function shall
 13420 place output on the standard output stream *stdout*. The *sprintf()* function shall place output
 13421 followed by the null byte, '\0', in consecutive bytes starting at **s*; it is the user's responsibility
 13422 to ensure that enough space is available.

13423 The *snprintf()* function shall be equivalent to *sprintf()*, with the addition of the *n* argument
 13424 which states the size of the buffer referred to by *s*. If *n* is zero, nothing shall be written and *s* may
 13425 be a null pointer. Otherwise, output bytes beyond the *n*-1st shall be discarded instead of being
 13426 written to the array, and a null byte is written at the end of the bytes actually written into the
 13427 array.

13428 If copying takes place between objects that overlap as a result of a call to *sprintf()* or *snprintf()*,
 13429 the results are undefined.

13430 Each of these functions converts, formats, and prints its arguments under control of the *format*.
 13431 The *format* is a character string, beginning and ending in its initial shift state, if any. The *format* is
 13432 composed of zero or more directives: *ordinary characters*, which are simply copied to the output
 13433 stream, and *conversion specifications*, each of which shall result in the fetching of zero or more
 13434 arguments. The results are undefined if there are insufficient arguments for the *format*. If the
 13435 *format* is exhausted while arguments remain, the excess arguments shall be evaluated but are
 13436 otherwise ignored.

13437 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than
 13438 to the next unused argument. In this case, the conversion specifier character % (see below) is
 13439 replaced by the sequence "%*n\$*", where *n* is a decimal integer in the range [1,{NL_ARGMAX}],
 13440 giving the position of the argument in the argument list. This feature provides for the definition
 13441 of format strings that select arguments in an order appropriate to specific languages (see the
 13442 EXAMPLES section).

13443 The *format* can contain either numbered argument conversion specifications (that is, "%*n\$*" and
 13444 "%*m\$*"), or unnumbered argument conversion specifications (that is, % and *), but not both. The
 13445 only exception to this is that %% can be mixed with the "%*n\$*" form. The results of mixing
 13446 numbered and unnumbered argument specifications in a *format* string are undefined. When
 13447 numbered argument specifications are used, specifying the *N*th argument requires that all the
 13448 leading arguments, from the first to the (*N*-1)th, are specified in the *format* string.

13449 In format strings containing the "%*n\$*" form of conversion specification, numbered arguments
 13450 in the argument list can be referenced from the *format* string as many times as required.

13451 In format strings containing the % form of conversion specification, each conversion specification
 13452 uses the first unused argument in the argument list.

13453 CX All forms of the *fprintf()* 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 *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.').

13457 XSI Each conversion specification is introduced by the '%' character or by the character sequence "%*n\$*", after which the following appear in sequence:

- 13459 • Zero or more *flags* (in any order), which modify the meaning of the conversion specification.
- 13460 • An optional minimum *field width*. 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 the left-adjustment flag ('-'), described below, is given to the field width. The field width takes the form of an asterisk ('*'), described below, or a decimal integer.
- 13464 • An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversion specifiers; the number of digits to appear after the radix character for the a, A, e, E, f, and F conversion specifiers; the maximum number of significant digits for the g and G conversion specifiers; or the maximum number of bytes to be printed from a string in the s and S conversion specifiers. The precision takes the form of a period ('.') followed either by an asterisk ('*'), described below, or an optional decimal digit string, where a null digit string is treated as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- 13472 • An optional length modifier that specifies the size of the argument.
- 13473 • A *conversion specifier* character that indicates the type of conversion to be applied.

13474 A field width, or precision, or both, may be indicated by an asterisk ('*'). In this case an argument of type **int** supplies the field width or precision. Applications shall ensure that arguments specifying field width, or precision, or both appear in that order before the argument, if any, to be converted. A negative field width is taken as a '-' flag followed by a positive field width. A negative precision is taken as if the precision were omitted. In format strings containing the "%*n\$*" form of a conversion specification, a field width or precision may be indicated by the sequence "**m\$*", where *m* is a decimal integer in the range [1,{NL_ARGMAX}] giving the position in the argument list (after the *format* argument) of an integer argument containing the field width or precision, for example:

```
13483 printf( "%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec );
```

13484 The flag characters and their meanings are:

- 13485 XSI
 - ' The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %F, %g, or %G) shall be formatted with thousands' grouping characters. For other conversions the behavior is undefined. The non-monetary grouping character is used.
 - The result of the conversion shall be left-justified within the field. The conversion is right-justified if this flag is not specified.
 - + The result of a signed conversion shall always begin with a sign ('+' or '-'). The conversion shall begin with a sign only when a negative value is converted if this flag is not specified.
 - <space> If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a <space> shall be prefixed to the result. This means that if the <space> and '+' flags both appear, the <space> flag shall be ignored.
 - # Specifies that the value is to be converted to an alternative form. For o conversion, it increases the precision (if necessary) to force the first digit of the result to be zero. For x

13498 or X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A, e, E, f, F, g, and G conversion specifiers, the result shall always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversion specifiers, trailing zeros shall *not* be removed from the result as they normally are. For other conversion specifiers, the behavior is undefined.

13504 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 is ignored. For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag is ignored. ~~If the '0' and ' ' flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined.~~

13510 The length modifiers and their meanings are:

13511 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.

13516 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.

13521 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.

13526 ll (ell-ell) 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.

13530 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.

13533 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.

13536 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.

13539 L Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a **long double** argument.

13541 If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

13543	The conversion specifiers and their meanings are:	
13544	d, i	The int argument shall be converted to a signed decimal in the style "[-]ddd". 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 is 1. The result of converting zero with an explicit precision of zero shall be no characters.
13549	o	The unsigned argument shall be converted to unsigned octal format in the style "ddd". 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 is 1. The result of converting zero with an explicit precision of zero shall be no characters.
13554	u	The unsigned argument shall be converted to unsigned decimal format in the style "ddd". 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 is 1. The result of converting zero with an explicit precision of zero shall be no characters.
13559	x	The unsigned argument shall be converted to unsigned hexadecimal format in the style "ddd"; 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 is 1. The result of converting zero with an explicit precision of zero shall be no characters.
13564	X	Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead of "abcdef".
13566	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 is 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 appears before it. The low-order digit shall be rounded in an implementation-defined manner.
13572		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(<i>n-char-sequence</i>)" or "[-]nan"; 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.
13578	e, E	The double argument shall be converted in the style "[-]d.ddde \pm dd", where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is 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 low-order digit shall be rounded in an implementation-defined manner. The E conversion specifier 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.
13586		A double argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.
13588	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

13590 digits. If an explicit precision is zero, it shall be taken as 1. The style used depends on
13591 the value converted; style e (or E) shall be used only if the exponent resulting from
13592 such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros
13593 shall be removed from the fractional portion of the result; a radix character shall appear
13594 only if it is followed by a digit or a '#' flag is present.

13595 A **double** argument representing an infinity or NaN shall be converted in the style of
13596 an f or F conversion specifier.

13597 a, A
13598 A **double** argument representing a floating-point number shall be converted in the style
13599 "[-]0xh.hhhhp±d", where there is one hexadecimal digit (which shall be non-
13600 zero if the argument is a normalized floating-point number and is otherwise un-
13601 specified) before the decimal-point character and the number of hexadecimal digits
13602 after it is equal to the precision; if the precision is missing and FLT_RADIX is a power
13603 of 2, then the precision shall be sufficient for an exact representation of the value; if the
13604 precision is missing and FLT_RADIX is not a power of 2, then the precision shall be
13605 sufficient to distinguish values of type **double**, except that trailing zeros may be
13606 omitted; if the precision is zero and the '#' flag is not specified, no decimal-point
13607 character shall appear. The letters "abcdef" shall be used for a conversion and the
13608 letters "ABCDEF" for A conversion. The A conversion specifier produces a number with
13609 'x' and 'p' instead of 'x' and 'p'. The exponent shall always contain at least one
13610 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.

13611 A **double** argument representing an infinity or NaN shall be converted in the style of
13612 an f or F conversion specifier.

13613 c
13614 The **int** argument shall be converted to an **unsigned char**, and the resulting byte shall
be written.

13615 If an l (ell) qualifier is present, the **wint_t** argument shall be converted as if by an ls
13616 conversion specification with no precision and an argument that points to a two-
13617 element array of type **wchar_t**, the first element of which contains the **wint_t** argument
13618 to the ls conversion specification and the second element contains a null wide
13619 character.

13620 s
13621 The argument shall be a pointer to an array of **char**. Bytes from the array shall be
13622 written up to (but not including) any terminating null byte. If the precision is specified,
13623 no more than that many bytes shall be written. If the precision is not specified or is
13624 greater than the size of the array, the application shall ensure that the array contains a
null byte.

13625 If an l (ell) qualifier is present, the argument shall be a pointer to an array of type
13626 **wchar_t**. Wide characters from the array shall be converted to characters (each as if by
13627 a call to the **wcrtomb()** function, with the conversion state described by an **mbstate_t**
13628 object initialized to zero before the first wide character is converted) up to and
13629 including a terminating null wide character. The resulting characters shall be written
13630 up to (but not including) the terminating null character (byte). If no precision is
13631 specified, the application shall ensure that the array contains a null wide character. If a
13632 precision is specified, no more than that many characters (bytes) shall be written
13633 (including shift sequences, if any), and the array shall contain a null wide character if,
13634 to equal the character sequence length given by the precision, the function would need
13635 to access a wide character one past the end of the array. In no case shall a partial
13636 character be written.

13637	p	The argument shall be a pointer to void . The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.
13639	n	The argument shall be a pointer to an integer into which is written the number of bytes written to the output so far by this call to one of the <i>fprintf()</i> functions. No argument is converted.
13642 XSI	C	Equivalent to <i>lc</i> .
13643 XSI	S	Equivalent to <i>ls</i> .
13644	%	Print a '%' character; no argument is converted. The complete conversion specification shall be %%.
13645		If a conversion specification does not match one of the above forms, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.
13646		In no case shall a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field shall be expanded to contain the conversion result. Characters generated by <i>fprintf()</i> and <i>printf()</i> are printed as if <i>fputc()</i> had been called.
13647		For the a and A conversion specifiers, if FLT_RADIX is a power of 2, the value shall be correctly rounded to a hexadecimal floating number with the given precision.
13648		For a and A conversions, if FLT_RADIX is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.
13649		For the e, E, f, F, g, and G conversion specifiers, if the number of significant decimal digits is at most DECIMAL_DIG, then the result should be correctly rounded. If the number of significant decimal digits is more than DECIMAL_DIG but the source value is exactly representable with DECIMAL_DIG digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings L < U, both having DECIMAL_DIG significant digits; the value of the resultant decimal string D should satisfy L <= D <= U, with the extra stipulation that the error should have a correct sign for the current rounding direction.
13650		
13651		
13652		
13653		
13654		
13655		
13656		
13657		
13658		
13659		
13660		
13661		
13662		
13663		
13664		
13665		
13666 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 successful execution of <i>fprintf()</i> or <i>printf()</i> and the next successful completion of a call to <i>fflush()</i> or <i>fclose()</i> on the same stream or a call to <i>exit()</i> or <i>abort()</i> .
13667		
13668		

13669 RETURN VALUE

13670	Upon successful completion, the <i>fprintf()</i> and <i>printf()</i> functions shall return the number of bytes transmitted.
13671	
13672	Upon successful completion, the <i>sprintf()</i> function shall return the number of bytes written to <i>s</i> , excluding the terminating null byte.
13673	
13674	Upon successful completion, the <i>snprintf()</i> function shall return the number of bytes that would be written to <i>s</i> had <i>n</i> been sufficiently large excluding the terminating null byte.
13675	
13676	If an output error was encountered, these functions shall return a negative value.
13677	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 would have been written had <i>n</i> been sufficiently large excluding the terminating null shall be returned, and <i>s</i> may be a null pointer.
13678	
13679	

13680 ERRORS

13681 For the conditions under which *fprintf()* and *printf()* fail and may fail, refer to *fputc()* or
13682 *fputwc()*.

13683 In addition, all forms of *fprintf()* may fail if:

13684 XSI [EILSEQ] A wide-character code that does not correspond to a valid character has been
13685 detected.

13686 XSI [EINVAL] There are insufficient arguments.

13687 The *printf()* and *fprintf()* functions may fail if:

13688 XSI [ENOMEM] Insufficient storage space is available.

13689 The *snprintf()* function shall fail if:

13690 XSI [EOVERFLOW] The value of *n* is greater than {INT_MAX} or the number of bytes needed to
13691 hold the output excluding the terminating null is greater than {INT_MAX}.

13692 EXAMPLES

13693 **Printing Language-Independent Date and Time**

13694 The following statement can be used to print date and time using a language-independent
13695 format:

13696 `printf(format, weekday, month, day, hour, min);`

13697 For American usage, *format* could be a pointer to the following string:

13698 `"%s, %s %d, %d:%.2d\n"`

13699 This example would produce the following message:

13700 `Sunday, July 3, 10:02`

13701 For German usage, *format* could be a pointer to the following string:

13702 `"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"`

13703 This definition of *format* would produce the following message:

13704 `Sonntag, 3. Juli, 10:02`

13705 **Printing File Information**

13706 The following example prints information about the type, permissions, and number of links of a
13707 specific file in a directory.

13708 The first two calls to *printf()* use data decoded from a previous *stat()* call. The user-defined
13709 *strperm()* function shall return a string similar to the one at the beginning of the output for the
13710 following command:

13711 `ls -l`

13712 The next call to *printf()* outputs the owner's name if it is found using *getpwuid()*; the *getpwuid()*
13713 function shall return a **passwd** structure from which the name of the user is extracted. If the user
13714 name is not found, the program instead prints out the numeric value of the user ID.

13715 The next call prints out the group name if it is found using *getgrgid()*; *getgrgid()* is very similar to
13716 *getpwuid()* except that it shall return group information based on the group number. Once
13717 again, if the group is not found, the program prints the numeric value of the group for the entry.

```
13718     The final call to printf() prints the size of the file.  
13719         #include <stdio.h>  
13720         #include <sys/types.h>  
13721         #include <pwd.h>  
13722         #include <grp.h>  
13723  
13724             char *strperm (mode_t);  
13725             ...  
13726             struct stat statbuf;  
13727             struct passwd *pwd;  
13728             struct group *grp;  
13729             ...  
13730             printf("%10.10s", strperm (statbuf.st_mode));  
13731             printf("%4d", statbuf.st_nlink);  
13732  
13733             if ((pwd = getpwuid(statbuf.st_uid)) != NULL)  
13734                 printf(" %-8.8s", pwd->pw_name);  
13735             else  
13736                 printf(" %-8ld", (long) statbuf.st_uid);  
13737  
13738             if ((grp = getgrgid(statbuf.st_gid)) != NULL)  
13739                 printf(" %-8.8s", grp->gr_name);  
13740             else  
13741                 printf(" %-8ld", (long) statbuf.st_gid);  
13742             printf("%9jd", (intmax_t) statbuf.st_size);  
13743             ...  
13744  
13745  
13746
```

Printing a Localized Date String

13742 The following example gets a localized date string. The *nl_langinfo()* function shall return the
13743 localized date string, which specifies the order and layout of the date. The *strftime()* function
13744 takes this information and, using the **tm** structure for values, places the date and time
13745 information into *datestring*. The *printf()* function then outputs *datestring* and the name of the
13746 entry.

```
13747         #include <stdio.h>  
13748         #include <time.h>  
13749         #include <langinfo.h>  
13750         ...  
13751         struct dirent *dp;  
13752         struct tm *tm;  
13753         char datestring[256];  
13754         ...  
13755         strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);  
13756         printf(" %s %s\n", datestring, dp->d_name);  
13757         ...  
13758
```

13758 **Printing Error Information**

13759 The following example uses *fprintf()* to write error information to standard error.

13760 In the first group of calls, the program tries to open the password lock file named **LOCKFILE**. If
13761 the file already exists, this is an error, as indicated by the **O_EXCL** flag on the *open()* function. If
13762 the call fails, the program assumes that someone else is updating the password file, and the
13763 program exits.

13764 The next group of calls saves a new password file as the current password file by creating a link
13765 between **LOCKFILE** and the new password file **PASSWDFILE**.

```
13766 #include <sys/types.h>
13767 #include <sys/stat.h>
13768 #include <fcntl.h>
13769 #include <stdio.h>
13770 #include <stdlib.h>
13771 #include <unistd.h>
13772 #include <string.h>
13773 #include <errno.h>

13774 #define LOCKFILE "/etc/ptmp"
13775 #define PASSWDFILE "/etc/passwd"
13776 ...
13777 int pfd;
13778 ...
13779 if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
13780     S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
13781 {
13782     fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
13783     exit(1);
13784 }
13785 ...
13786 if (link(LOCKFILE,PASSWDFILE) == -1) {
13787     fprintf(stderr, "Link error: %s\n", strerror(errno));
13788     exit(1);
13789 }
13790 ...
```

13791 **Printing Usage Information**

13792 The following example checks to make sure the program has the necessary arguments, and uses
13793 *fprintf()* to print usage information if the expected number of arguments is not present.

```
13794 #include <stdio.h>
13795 #include <stdlib.h>
13796 ...
13797 char *Options = "hdbtl";
13798 ...
13799 if (argc < 2) {
13800     fprintf(stderr, "Usage: %s -%s <file>\n", argv[0], Options); exit(1);
13801 }
13802 ...
```

13803 **Formatting a Decimal String**

13804 The following example prints a key and data pair on *stdout*. Note use of the '*' (asterisk) in the
13805 format string; this ensures the correct number of decimal places for the element based on the
13806 number of elements requested.

```
13807 #include <stdio.h>
13808 ...
13809 long i;
13810 char *keystr;
13811 int elementlen, len;
13812 ...
13813 while (len < elementlen) {
13814 ...
13815     printf("%s Element%0*ld\n", keystr, elementlen, i);
13816 ...
13817 }
```

13818 **Creating a Filename**

13819 The following example creates a filename using information from a previous *getpwnam()*
13820 function that returned the HOME directory of the user.

```
13821 #include <stdio.h>
13822 #include <sys/types.h>
13823 #include <unistd.h>
13824 ...
13825 char filename[PATH_MAX+1];
13826 struct passwd *pw;
13827 ...
13828 sprintf(filename, "%s/%d.out", pw->pw_dir, getpid());
13829 ...
```

13830 **Reporting an Event**

13831 The following example loops until an event has timed out. The *pause()* function waits forever
13832 unless it receives a signal. The *fprintf()* statement should never occur due to the possible return
13833 values of *pause()*.

```
13834 #include <stdio.h>
13835 #include <unistd.h>
13836 #include <string.h>
13837 #include <errno.h>
13838 ...
13839 while (!event_complete) {
13840 ...
13841     if (pause() != -1 || errno != EINTR)
13842         fprintf(stderr, "pause: unknown error: %s\n", strerror(errno));
13843 }
13844 ...
```

13845 **Printing Monetary Information**

13846 The following example uses *strfmon()* to convert a number and store it as a formatted monetary
 13847 string named *convbuf*. If the first number is printed, the program prints the format and the
 13848 description; otherwise, it just prints the number.

```

13849     #include <monetary.h>
13850     #include <stdio.h>
13851     ...
13852     struct tblfmt {
13853         char *format;
13854         char *description;
13855     };
13856
13857     struct tblfmt table[] = {
13858         { "%n", "default formatting" },
13859         { "%11n", "right align within an 11 character field" },
13860         { "%#5n", "aligned columns for values up to 99 999" },
13861         { "%-*#5n", "specify a fill character" },
13862         { "%=0#5n", "fill characters do not use grouping" },
13863         { "%^#5n", "disable the grouping separator" },
13864         { "%^#5.0n", "round off to whole units" },
13865         { "%^#5.4n", "increase the precision" },
13866         { "%(#5n", "use an alternative pos/neg style" },
13867         { "%!(#5n", "disable the currency symbol" },
13868     };
13869     ...
13870     float input[3];
13871     int i, j;
13872     char convbuf[100];
13873     ...
13874     strfmon(convbuf, sizeof(convbuf), table[i].format, input[j]);
13875     if (j == 0) {
13876         printf("%s%s%s\n", table[i].format,
13877               convbuf, table[i].description);
13878     }
13879     else {
13880         printf("%s\n", convbuf);
13881     }
13882     ...
  
```

13882 **Printing Wide Characters**

13883 The following example prints a series of wide characters. Suppose that "L'@'" expands to three
 13884 bytes:

```

13885     wchar_t wz [3] = L"@@";           // Zero-terminated
13886     wchar_t wn [3] = L"@@@";          // Untermminated
13887
13888     fprintf (stdout,"%ls", wz);      // Outputs 6 bytes
13889     fprintf (stdout,"%ls", wn);      // Undefined because wn has no terminator
13890     fprintf (stdout,"%4ls", wz);      // Outputs 3 bytes
13891     fprintf (stdout,"%4ls", wn);      // Outputs 3 bytes; no terminator needed
13892     fprintf (stdout,"%9ls", wz);      // Outputs 6 bytes
  
```

13892 `fprintf (stdout,"%9ls", wn); // Outputs 9 bytes; no terminator needed`
13893 `fprintf (stdout,"%10ls", wz); // Outputs 6 bytes`
13894 `fprintf (stdout,"%10ls", wn); // Undefined because wn has no terminator`

13895 In the last line of the example, after processing three characters, nine bytes have been output.
13896 The fourth character must then be examined to determine whether it converts to one byte or
13897 more. If it converts to more than one byte, the output is only nine bytes. Since there is no fourth
13898 character in the array, the behavior is undefined.

13899 APPLICATION USAGE

13900 If the application calling *fprintf()* has any objects of type **wint_t** or **wchar_t**, it must also include
13901 the <**wchar.h**> header to have these objects defined.

13902 RATIONALE

13903 None.

13904 FUTURE DIRECTIONS

13905 None.

13906 SEE ALSO

13907 *fputc()*, *fscanf()*, *setlocale()*, *strfmon()*, *wcrtomb()*, the Base Definitions volume of
13908 IEEE Std 1003.1-2001, Chapter 7, Locale, <**stdio.h**>, <**wchar.h**>

13909 CHANGE HISTORY

13910 First released in Issue 1. Derived from Issue 1 of the SVID.

13911 Issue 5

13912 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the **l** (ell) qualifier can
13913 now be used with **c** and **s** conversion specifiers.

13914 The *snprintf()* function is new in Issue 5.

13915 Issue 6

13916 Extensions beyond the ISO C standard are marked.

13917 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

13918 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 13919 • The prototypes for *fprintf()*, *printf()*, *snprintf()*, and *sprintf()* are updated, and the XSI
13920 shading is removed from *snprintf()*.
- 13921 • The description of *snprintf()* is aligned with the ISO C standard. Note that this supersedes
13922 the *snprintf()* description in The Open Group Base Resolution bwg98-006, which changed the
13923 behavior from Issue 5.
- 13924 • The DESCRIPTION is updated.

13925 The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion
13926 specification” consistently.

13927 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

13928 An example of printing wide characters is added.

13929 NAME

13930 fputc — put a byte on a stream

13931 SYNOPSIS

13932 #include <stdio.h>

13933 int fputc(int *c*, FILE **stream*);

13934 DESCRIPTION

13935 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 13936 conflict between the requirements described here and the ISO C standard is unintentional. This
 13937 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

13938 The *fputc()* function shall write the byte specified by *c* (converted to an **unsigned char**) to the
 13939 output stream pointed to by *stream*, at the position indicated by the associated file-position
 13940 indicator for the stream (if defined), and shall advance the indicator appropriately. If the file
 13941 cannot support positioning requests, or if the stream was opened with append mode, the byte
 13942 shall be appended to the output stream.

13943 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
 13944 execution of *fputc()* and the next successful completion of a call to *fflush()* or *fclose()* on the same
 13945 stream or a call to *exit()* or *abort()*.

13946 RETURN VALUE

13947 Upon successful completion, *fputc()* shall return the value it has written. Otherwise, it shall
 13948 CX return EOF, the error indicator for the stream shall be set, and *errno* shall be set to indicate the
 13949 error.

13950 ERRORS

13951 The *fputc()* function shall fail if either the *stream* is unbuffered or the *stream*'s buffer needs to be
 13952 flushed, and:

13953 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the
 13954 thread would be delayed in the write operation. 2

13955 CX [EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for
 13956 writing.

13957 CX [EFBIG] An attempt was made to write to a file that exceeds the maximum file size.

13958 XSI [EFBIG] An attempt was made to write to a file that exceeds the process' file size limit.

13959 CX [EFBIG] The file is a regular file and an attempt was made to write at or beyond the
 13960 offset maximum.

13961 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data
 13962 was transferred.

13963 CX [EIO] A physical I/O error has occurred, or the process is a member of a
 13964 background process group attempting to write to its controlling terminal,
 13965 TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the
 13966 process group of the process is orphaned. This error may also be returned
 13967 under implementation-defined conditions.

13968 CX [ENOSPC] There was no free space remaining on the device containing the file.

13969 CX [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by
 13970 any process. A SIGPIPE signal shall also be sent to the thread.

13971 The *fputc()* function may fail if:

13972 CX [ENOMEM] Insufficient storage space is available.

13973 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the
13974 capabilities of the device.

13975 EXAMPLES

13976 None.

13977 APPLICATION USAGE

13978 None.

13979 RATIONALE

13980 None.

13981 FUTURE DIRECTIONS

13982 None.

13983 SEE ALSO

13984 *ferror()*, *fopen()*, *getrlimit()*, *putc()*, *puts()*, *setbuf()*, *ulimit()*, the Base Definitions volume of
13985 IEEE Std 1003.1-2001, <stdio.h>

13986 CHANGE HISTORY

13987 First released in Issue 1. Derived from Issue 1 of the SVID.

13988 Issue 5

13989 Large File Summit extensions are added.

13990 Issue 6

13991 Extensions beyond the ISO C standard are marked.

13992 The following new requirements on POSIX implementations derive from alignment with the
13993 Single UNIX Specification:

- The [EIO] and [EFBIG] mandatory error conditions are added.

- The [ENOMEM] and [ENXIO] optional error conditions are added.

13996 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/37 is applied, updating the [EAGAIN] 2
13997 error in the ERRRORS section from ‘the process would be delayed’ to ‘the thread would be 2
13998 delayed’.

13999 NAME

14000 *fputs* — put a string on a stream

14001 SYNOPSIS

```
14002        #include <stdio.h>
14003        int fputs(const char *restrict s, FILE *restrict stream);
```

14004 DESCRIPTION

14005 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14006 conflict between the requirements described here and the ISO C standard is unintentional. This
14007 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

14008 The *fputs()* function shall write the null-terminated string pointed to by *s* to the stream pointed
14009 to by *stream*. The terminating null byte shall not be written.

14010 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
14011 execution of *fputs()* and the next successful completion of a call to *fflush()* or *fclose()* on the same
14012 stream or a call to *exit()* or *abort()*.

14013 RETURN VALUE

14014 Upon successful completion, *fputs()* shall return a non-negative number. Otherwise, it shall
14015 CX return EOF, set an error indicator for the stream, and set *errno* to indicate the error.

14016 ERRORS

14017 Refer to *fputc()*.

14018 EXAMPLES**14019 Printing to Standard Output**

14020 The following example gets the current time, converts it to a string using *localtime()* and
14021 *asctime()*, and prints it to standard output using *fputs()*. It then prints the number of minutes to
14022 an event for which it is waiting.

```
14023        #include <time.h>
14024        #include <stdio.h>
14025        ...
14026        time_t now;
14027        int minutes_to_event;
14028        ...
14029        time(&now);
14030        printf("The time is ");
14031        fputs(asctime(localtime(&now)), stdout);
14032        printf("There are still %d minutes to the event.\n",
14033            minutes_to_event);
14034        ...
```

14035 APPLICATION USAGE

14036 The *puts()* function appends a <newline> while *fputs()* does not.

14037 RATIONALE

14038 None.

14039 FUTURE DIRECTIONS

14040 None.

14041 **SEE ALSO**

14042 *fopen()*, *putc()*, *puts()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

14043 **CHANGE HISTORY**

14044 First released in Issue 1. Derived from Issue 1 of the SVID.

14045 **Issue 6**

14046 Extensions beyond the ISO C standard are marked.

14047 The *fputs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

14048 NAME

14049 fputwc — put a wide-character code on a stream

14050 SYNOPSIS

```
14051 #include <stdio.h>
14052 #include <wchar.h>
14053 wint_t fputwc(wchar_t wc, FILE *stream);
```

14054 DESCRIPTION

14055 CX 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.

14058 The *fputwc()* function shall write the character corresponding to the wide-character code *wc* to the output stream pointed to by *stream*, at the position indicated by the associated file-position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream. If an error occurs while writing the character, the shift state of the output file is left in an undefined state.

14064 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful execution of *fputwc()* and the next successful completion of a call to *fflush()* or *fclose()* on the same stream or a call to *exit()* or *abort()*.

14067 RETURN VALUE

14068 Upon successful completion, *fputwc()* shall return *wc*. Otherwise, it shall return WEOF, the error indicator for the stream shall be set, and *errno* shall be set to indicate the error.

14069 CX ERRORS

14071 The *fputwc()* function shall fail if either the stream is unbuffered or data in the *stream*'s buffer needs to be written, and:

14073 CX [EAGAIN]	The <i>O_NONBLOCK</i> flag is set for the file descriptor underlying <i>stream</i> and the thread would be delayed in the write operation.	2
14075 CX [EBADF]	The file descriptor underlying <i>stream</i> is not a valid file descriptor open for writing.	
14077 CX [EFBIG]	An attempt was made to write to a file that exceeds the maximum file size or the process' file size limit.	
14079 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.	
14081 [EILSEQ]	The wide-character code <i>wc</i> does not correspond to a valid character.	
14082 CX [EINTR]	The write operation was terminated due to the receipt of a signal, and no data was transferred.	
14084 CX [EIO]	A physical I/O error has occurred, or the process is a member of a background process group attempting to write 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.	
14089 CX [ENOSPC]	There was no free space remaining on the device containing the file.	
14090 CX [EPIPE]	An attempt is 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.	

14092	The <i>fputwc()</i> function may fail if:	
14093 CX	[ENOMEM]	Insufficient storage space is available.
14094 CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the
14095		capabilities of the device.

14096 EXAMPLES

14097 None.

14098 APPLICATION USAGE

14099 None.

14100 RATIONALE

14101 None.

14102 FUTURE DIRECTIONS

14103 None.

14104 SEE ALSO

14105 *ferror()*, *fopen()*, *setbuf()*, *ulimit()*, the Base Definitions volume of IEEE Std 1003.1-2001,
14106 <**stdio.h**>, <**wchar.h**>

14107 CHANGE HISTORY

14108 First released in Issue 4. Derived from the MSE working draft.

14109 Issue 5

14110 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc*
14111 is changed from *wint_t* to *wchar_t*.

14112 The Optional Header (OH) marking is removed from <**stdio.h**>.

14113 Large File Summit extensions are added.

14114 Issue 6

14115 Extensions beyond the ISO C standard are marked.

14116 The following new requirements on POSIX implementations derive from alignment with the
14117 Single UNIX Specification:

- The [EFBIG] and [EIO] mandatory error conditions are added.

- The [ENOMEM] and [ENXIO] optional error conditions are added.

14118 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/38 is applied, updating the [EAGAIN] 2
14119 error in the ERRORS section from “the process would be delayed” to “the thread would be 2
14120 delayed”. 2
14121
14122

14123 NAME

14124 *fputws* — put a wide-character string on a stream

14125 SYNOPSIS

```
14126     #include <stdio.h>
14127     #include <wchar.h>
14128     int fputws(const wchar_t *restrict ws, FILE *restrict stream);
```

14129 DESCRIPTION

14130 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14131 conflict between the requirements described here and the ISO C standard is unintentional. This
14132 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

14133 The *fputws()* function shall write a character string corresponding to the (null-terminated)
14134 wide-character string pointed to by *ws* to the stream pointed to by *stream*. No character
14135 corresponding to the terminating null wide-character code shall be written.

14136 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful
14137 execution of *fputws()* and the next successful completion of a call to *fflush()* or *fclose()* on the
14138 same stream or a call to *exit()* or *abort()*.

14139 RETURN VALUE

14140 Upon successful completion, *fputws()* shall return a non-negative number. Otherwise, it shall
14141 CX return -1, set an error indicator for the stream, and set *errno* to indicate the error.

14142 ERRORS

14143 Refer to *fputwc()*.

14144 EXAMPLES

14145 None.

14146 APPLICATION USAGE

14147 The *fputws()* function does not append a <newline>.

14148 RATIONALE

14149 None.

14150 FUTURE DIRECTIONS

14151 None.

14152 SEE ALSO

14153 *open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>, <wchar.h>

14154 CHANGE HISTORY

14155 First released in Issue 4. Derived from the MSE working draft.

14156 Issue 5

14157 The Optional Header (OH) marking is removed from <stdio.h>.

14158 Issue 6

14159 Extensions beyond the ISO C standard are marked.

14160 The *fputws()* prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

14161 NAME

14162 fread — binary input

14163 SYNOPSIS

```
14164     #include <stdio.h>
14165
14166     size_t fread(void *restrict ptr, size_t size, size_t nitems,
14167                   FILE *restrict stream);
```

14167 DESCRIPTION

14168 CX 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.

14171 The *fread()* function shall read into the array pointed to by *ptr* up to *nitems* elements whose size is specified by *size* in bytes, from the stream pointed to by *stream*. For each object, *size* calls shall be made to the *fgetc()* function and the results stored, in the order read, in an array of **unsigned char** exactly overlaying the object. The file position indicator for the stream (if defined) shall be advanced by the number of bytes successfully read. If an error occurs, the resulting value of the file position indicator for the stream is unspecified. If a partial element is read, its value is unspecified.

14178 CX The *fread()* function may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*, *fgetwc()*, *fgetws()*, *fread()*, *fscanf()*, *getc()*, *getchar()*, *gets()*, or *scanf()* using *stream* that returns data not supplied by a prior call to *ungetc()* or *ungetwc()*.

14182 RETURN VALUE

14183 Upon successful completion, *fread()* shall return the number of elements successfully read which is less than *nitems* only if a read error or end-of-file is encountered. If *size* or *nitems* is 0, *fread()* shall return 0 and the contents of the array and the state of the stream remain unchanged. 14186 CX Otherwise, if a read error occurs, the error indicator for the stream shall be set, and *errno* shall be set to indicate the error.

14188 ERRORS

14189 Refer to *fgetc()*.

14190 EXAMPLES

14191 Reading from a Stream

14192 The following example reads a single element from the *fp* stream into the array pointed to by *buf*.

```
14193     #include <stdio.h>
14194
14195     ...
14196     size_t bytes_read;
14197     char buf[100];
14198     FILE *fp;
14199
14200     ...
14201     bytes_read = fread(buf, sizeof(buf), 1, fp);
14202
14203     ...
```

14201 APPLICATION USAGE

14202 The *ferror()* or *feof()* functions must be used to distinguish between an error condition and an end-of-file condition.

14204 Because of possible differences in element length and byte ordering, files written using *fwrite()* are application-dependent, and possibly cannot be read using *fread()* by a different application

14206 or by the same application on a different processor.

14207 **RATIONALE**

14208 None.

14209 **FUTURE DIRECTIONS**

14210 None.

14211 **SEE ALSO**

14212 *feof()*, *ferror()*, *fgetc()*, *fopen()*, *getc()*, *gets()*, *scanf()*, the Base Definitions volume of
14213 IEEE Std 1003.1-2001, <stdio.h>

14214 **CHANGE HISTORY**

14215 First released in Issue 1. Derived from Issue 1 of the SVID.

14216 **Issue 6**

14217 Extensions beyond the ISO C standard are marked.

14218 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 14219 • The *fread()* prototype is updated.
- 14220 • The DESCRIPTION is updated to describe how the bytes from a call to *fgetc()* are stored.

14221 NAME

14222 free — free allocated memory

14223 SYNOPSIS

```
14224 #include <stdlib.h>
14225 void free(void *ptr);
```

14226 DESCRIPTION

14227 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14228 conflict between the requirements described here and the ISO C standard is unintentional. This
14229 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

14230 The *free()* function shall cause the space pointed to by *ptr* to be deallocated; that is, made
14231 available for further allocation. If *ptr* is a null pointer, no action shall occur. Otherwise, if the
14232 ADV argument does not match a pointer earlier returned by the *calloc()*, *malloc()*, *posix_memalign()*,
14233 XSI *realloc()*, or *strdup()* function, or if the space has been deallocated by a call to *free()* or *realloc()*,
14234 the behavior is undefined.

14235 Any use of a pointer that refers to freed space results in undefined behavior.

14236 RETURN VALUE

14237 The *free()* function shall not return a value.

14238 ERRORS

14239 No errors are defined.

14240 EXAMPLES

14241 None.

14242 APPLICATION USAGE

14243 There is now no requirement for the implementation to support the inclusion of `<malloc.h>`.

14244 RATIONALE

14245 None.

14246 FUTURE DIRECTIONS

14247 None.

14248 SEE ALSO

14249 *calloc()*, *malloc()*, *realloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`

14250 CHANGE HISTORY

14251 First released in Issue 1. Derived from Issue 1 of the SVID.

14252 Issue 6

14253 Reference to the *valloc()* function is removed.

14254 NAME

14255 freeaddrinfo, getaddrinfo — get address information

14256 SYNOPSIS

```

14257 #include <sys/socket.h>
14258 #include <netdb.h>
14259 void freeaddrinfo(struct addrinfo *ai);
14260 int getaddrinfo(const char *restrict nodename,
14261             const char *restrict servname,
14262             const struct addrinfo *restrict hints,
14263             struct addrinfo **restrict res);

```

14264 DESCRIPTION

14265 The **freeaddrinfo()** function shall free one or more **addrinfo** structures returned by **getaddrinfo()**,
 14266 along with any additional storage associated with those structures. If the *ai_next* field of the
 14267 structure is not null, the entire list of structures shall be freed. The **freeaddrinfo()** function shall
 14268 support the freeing of arbitrary sublists of an **addrinfo** list originally returned by **getaddrinfo()**.

14269 The **getaddrinfo()** function shall translate the name of a service location (for example, a host
 14270 name) and/or a service name and shall return a set of socket addresses and associated
 14271 information to be used in creating a socket with which to address the specified service.

14272 **Note:** In many cases it is implemented by the Domain Name System, as documented in RFC 1034, 1
 14273 RFC 1035, and RFC 1886. 1

14274 The **freeaddrinfo()** and **getaddrinfo()** functions shall be thread-safe.

14275 The *nodename* and *servname* arguments are either null pointers or pointers to null-terminated
 14276 strings. One or both of these two arguments shall be supplied by the application as a non-null
 14277 pointer.

14278 The format of a valid name depends on the address family or families. If a specific family is not
 14279 given and the name could be interpreted as valid within multiple supported families, the
 14280 implementation shall attempt to resolve the name in all supported families and, in absence of
 14281 errors, one or more results shall be returned.

14282 If the *nodename* argument is not null, it can be a descriptive name or can be an address string. If
 14283 IP6 the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, valid descriptive names
 14284 include host names. If the specified address family is AF_INET or AF_UNSPEC, address strings
 14285 using Internet standard dot notation as specified in *inet_addr()* are valid.

14286 IP6 If the specified address family is AF_INET6 or AF_UNSPEC, standard IPv6 text forms described
 14287 in *inet_ntop()* are valid.

14288 If *nodename* is not null, the requested service location is named by *nodename*; otherwise, the
 14289 requested service location is local to the caller.

14290 If *servname* is null, the call shall return network-level addresses for the specified *nodename*. If
 14291 *servname* is not null, it is a null-terminated character string identifying the requested service. This
 14292 can be either a descriptive name or a numeric representation suitable for use with the address
 14293 IP6 family or families. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, the
 14294 service can be specified as a string specifying a decimal port number.

14295 If the *hints* argument is not null, it refers to a structure containing input values that may direct
 14296 the operation by providing options and by limiting the returned information to a specific socket
 14297 type, address family, and/or protocol. In this *hints* structure every member other than *ai_flags*,
 14298 *ai_family*, *ai_socktype*, and *ai_protocol* shall be set to zero or a null pointer. A value of
 14299 AF_UNSPEC for *ai_family* means that the caller shall accept any address family. A value of zero

14300	for <i>ai_socktype</i> means that the caller shall accept any socket type. A value of zero for <i>ai_protocol</i>	
14301	means that the caller shall accept any protocol. If <i>hints</i> is a null pointer, the behavior shall be as if	
14302	it referred to a structure containing the value zero for the <i>ai_flags</i> , <i>ai_socktype</i> , and <i>ai_protocol</i>	
14303	fields, and AF_UNSPEC for the <i>ai_family</i> field.	
14304	The <i>ai_flags</i> field to which the <i>hints</i> parameter points shall be set to zero or be the bitwise-	1
14305	inclusive OR of one or more of the values AI_PASSIVE, AI_CANONNAME,	
14306	AI_NUMERICHOST, AI_NUMERICSERV, AI_V4MAPPED, AI_ALL, and AI_ADDRCONFIG.	1
14307	If the AI_PASSIVE flag is specified, the returned address information shall be suitable for use in	
14308	binding a socket for accepting incoming connections for the specified service. In this case, if the	
14309	<i>nodename</i> argument is null, then the IP address portion of the socket address structure shall be	
14310	set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the	
14311	AI_PASSIVE flag is not specified, the returned address information shall be suitable for a call to	
14312	<i>connect()</i> (for a connection-mode protocol) or for a call to <i>connect()</i> , <i>sendto()</i> , or <i>sendmsg()</i> (for a	
14313	connectionless protocol). In this case, if the <i>nodename</i> argument is null, then the IP address	
14314	portion of the socket address structure shall be set to the loopback address. The AI_PASSIVE	1
14315	flag shall be ignored if the <i>nodename</i> argument is not null.	1
14316	If the AI_CANONNAME flag is specified and the <i>nodename</i> argument is not null, the function	
14317	shall attempt to determine the canonical name corresponding to <i>nodename</i> (for example, if	
14318	<i>nodename</i> is an alias or shorthand notation for a complete name).	
14319	Note: Since different implementations use different conceptual models, the terms “canonical name”	1
14320	and “alias” cannot be precisely defined for the general case. However, Domain Name System	1
14321	implementations are expected to interpret them as they are used in RFC 1034.	1
14322	A numeric host address string is not a “name”, and thus does not have a “canonical name”	1
14323	form; no address to host name translation is performed. See below for handling of the case	1
14324	where a canonical name cannot be obtained.	1
14325	If the AI_NUMERICHOST flag is specified, then a non-null <i>nodename</i> string supplied shall be a	
14326	numeric host address string. Otherwise, an [EAI_NONAME] error is returned. This flag shall	
14327	prevent any type of name resolution service (for example, the DNS) from being invoked.	
14328	If the AI_NUMERICSERV flag is specified, then a non-null <i>servname</i> string supplied shall be a	
14329	numeric port string. Otherwise, an [EAI_NONAME] error shall be returned. This flag shall	
14330	prevent any type of name resolution service (for example, NIS+) from being invoked.	
14331 IP6	If the AI_V4MAPPED flag is specified along with an <i>ai_family</i> of AF_INET6, then <i>getaddrinfo()</i>	
14332	shall return IPv4-mapped IPv6 addresses on finding no matching IPv6 addresses (<i>ai_addrlen</i>	
14333	shall be 16). The AI_V4MAPPED flag shall be ignored unless <i>ai_family</i> equals AF_INET6. If the	
14334	AI_ALL flag is used with the AI_V4MAPPED flag, then <i>getaddrinfo()</i> shall return all matching	
14335	IPv6 and IPv4 addresses. The AI_ALL flag without the AI_V4MAPPED flag is ignored.	
14336 IP6	If the AI_ADDRCONFIG flag is specified, IPv4 addresses shall be returned only if an IPv4	1
14337 IP6	address is configured on the local system, and IPv6 addresses shall be returned only if an IPv6	1
14338	address is configured on the local system.	1
14339	The <i>ai_socktype</i> field to which argument <i>hints</i> points specifies the socket type for the service, as	
14340	defined in <i>socket()</i> . If a specific socket type is not given (for example, a value of zero) and the	
14341	service name could be interpreted as valid with multiple supported socket types, the	
14342	implementation shall attempt to resolve the service name for all supported socket types and, in	
14343	the absence of errors, all possible results shall be returned. A non-zero socket type value shall	
14344	limit the returned information to values with the specified socket type.	
14345	If the <i>ai_family</i> field to which <i>hints</i> points has the value AF_UNSPEC, addresses shall be	
14346	returned for use with any address family that can be used with the specified <i>nodename</i> and/or	
14347	<i>servname</i> . Otherwise, addresses shall be returned for use only with the specified address family.	

14348 If *ai_family* is not AF_UNSPEC and *ai_protocol* is not zero, then addresses are returned for use
14349 only with the specified address family and protocol; the value of *ai_protocol* shall be interpreted
14350 as in a call to the *socket()* function with the corresponding values of *ai_family* and *ai_protocol*.

14351 RETURN VALUE

14352 A zero return value for *getaddrinfo()* indicates successful completion; a non-zero return value
14353 indicates failure. The possible values for the failures are listed in the ERRORS section.

14354 Upon successful return of *getaddrinfo()*, the location to which *res* points shall refer to a linked list
14355 of **addrinfo** structures, each of which shall specify a socket address and information for use in
14356 creating a socket with which to use that socket address. The list shall include at least one
14357 **addrinfo** structure. The *ai_next* field of each structure contains a pointer to the next structure on
14358 the list, or a null pointer if it is the last structure on the list. Each structure on the list shall
14359 include values for use with a call to the *socket()* function, and a socket address for use with the
14360 *connect()* function or, if the AI_PASSIVE flag was specified, for use with the *bind()* function. The
14361 fields *ai_family*, *ai_socktype*, and *ai_protocol* shall be usable as the arguments to the *socket()*
14362 function to create a socket suitable for use with the returned address. The fields *ai_addr* and
14363 *ai_addrlen* are usable as the arguments to the *connect()* or *bind()* functions with such a socket,
14364 according to the AI_PASSIVE flag.

14365 If *nodename* is not null, and if requested by the AI_CANONNAME flag, the *ai_canonname* field of
14366 the first returned **addrinfo** structure shall point to a null-terminated string containing the
14367 canonical name corresponding to the input *nodename*; if the canonical name is not available, then
14368 *ai_canonname* shall refer to the *nodename* argument or a string with the same contents. The
14369 contents of the *ai_flags* field of the returned structures are undefined.

14370 All fields in socket address structures returned by *getaddrinfo()* that are not filled in through an
14371 explicit argument (for example, *sin6_flowinfo*) shall be set to zero.

14372 Note: This makes it easier to compare socket address structures.

14373 ERRORS

14374 The *getaddrinfo()* function shall fail and return the corresponding error value if: 2

14375 [EAI AGAIN] The name could not be resolved at this time. Future attempts may succeed.

14376 [EAI_BADFLAGS]

14377 The *flags* parameter had an invalid value.

14378 [EAI FAIL] A non-recoverable error occurred when attempting to resolve the name.

14379 [EAI_FAMILY] The address family was not recognized.

14380 [EAI_MEMORY] There was a memory allocation failure when trying to allocate storage for the
14381 return value.

14382 [EAI_NOMNAME] The name does not resolve for the supplied parameters.

14383 Neither *nodename* nor *servname* were supplied. At least one of these shall be
14384 supplied.

14385 [EAI_SERVICE] The service passed was not recognized for the specified socket type.

14386 [EAI_SOCKTYPE]

14387 The intended socket type was not recognized.

14388 [EAI_SYSTEM] A system error occurred; the error code can be found in *errno*.

14389 [EAI_OVERFLOW]

14390 An argument buffer overflowed.

14391 EXAMPLES

14392 None.

14393 APPLICATION USAGE

14394 If the caller handles only TCP and not UDP, for example, then the *ai_protocol* member of the *hints* structure should be set to IPPROTO_TCP when *getaddrinfo()* is called.

14396 If the caller handles only IPv4 and not IPv6, then the *ai_family* member of the *hints* structure should be set to AF_INET when *getaddrinfo()* is called.

14398 The term “canonical name” is misleading; it is taken from the Domain Name System (RFC 2181). 1
14399 It should be noted that the canonical name is a result of alias processing, and not necessarily a 1
14400 unique attribute of a host, address, or set of addresses. See RFC 2181 for more discussion of this 1
14401 in the Domain Name System context. 1

14402 RATIONALE

14403 None.

14404 FUTURE DIRECTIONS

14405 None.

14406 SEE ALSO

14407 *connect()*, *gai_strerror()*, *gethostbyaddr()*, *getnameinfo()*, *getservbyname()*, *socket()*, the Base
14408 Definitions volume of IEEE Std 1003.1-2001, <*netdb.h*>, <*sys/socket.h*>

14409 CHANGE HISTORY

14410 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

14411 The **restrict** keyword is added to the *getaddrinfo()* prototype for alignment with the 1
14412 ISO/IEC 9899:1999 standard. 1

14413 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/19 is applied, adding three notes to the 1
14414 DESCRIPTION and adding text to the APPLICATION USAGE related to the term “canonical 1
14415 name”. A reference to RFC 2181 is also added to the Informative References. 1

14416 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/20 is applied, making changes for 1
14417 alignment with IPv6. These include the following: 1

- 14418 • Adding AI_V4MAPPED, AI_ALL, and AI_ADDRCONFIG to the allowed values for the 1
14419 *ai_flags* field 1
- 14420 • Adding a description of AI_ADDRCONFIG 1
- 14421 • Adding a description of the consequences of ignoring the AI_PASSIVE flag. 1

14422 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/39 is applied, changing “corresponding 2
14423 value” to “corresponding error value” in the ERRORS section. 2

14424 NAME

14425 *freopen* — open a stream

14426 SYNOPSIS

```
14427        #include <stdio.h>
14428        FILE *freopen(const char *restrict filename, const char *restrict mode,
14429                    FILE *restrict stream);
```

14430 DESCRIPTION

14431 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 14432 conflict between the requirements described here and the ISO C standard is unintentional. This
 14433 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

14434 The *freopen()* function shall first attempt to flush the stream and close any file descriptor
 14435 associated with *stream*. Failure to flush or close the file descriptor successfully shall be ignored.
 14436 The error and end-of-file indicators for the stream shall be cleared.

14437 The *freopen()* function shall open the file whose pathname is the string pointed to by *filename* and
 14438 associate the stream pointed to by *stream* with it. The *mode* argument shall be used just as in
 14439 *fclose()*.

14440 The original stream shall be closed regardless of whether the subsequent open succeeds.

14441 If *filename* is a null pointer, the *freopen()* function shall attempt to change the mode of the stream
 14442 to that specified by *mode*, as if the name of the file currently associated with the stream had been
 14443 used. In this case, the file descriptor associated with the stream need not be closed if the call to
 14444 *freopen()* succeeds. It is implementation-defined which changes of mode are permitted (if any),
 14445 and under what circumstances.

14446 XSI After a successful call to the *freopen()* function, the orientation of the stream shall be cleared, the
 14447 encoding rule shall be cleared, and the associated **mbstate_t** object shall be set to describe an
 14448 initial conversion state.

14449 CX The largest value that can be represented correctly in an object of type **off_t** shall be established
 14450 as the offset maximum in the open file description.

14451 RETURN VALUE

14452 Upon successful completion, *freopen()* shall return the value of *stream*. Otherwise, a null pointer
 14453 CX shall be returned, and *errno* shall be set to indicate the error.

14454 ERRORS

14455 The *freopen()* function shall fail if:

14456 CX [EACCES] Search permission is denied on a component of the path prefix, or the file
 14457 exists and the permissions specified by *mode* are denied, or the file does not
 14458 exist and write permission is denied for the parent directory of the file to be
 14459 created.

14460 CX [EBADF] The file descriptor underlying the stream is not a valid file descriptor when
 14461 *filename* is a null pointer. 2

14462 CX [EINTR] A signal was caught during *freopen()*.

14463 CX [EISDIR] The named file is a directory and *mode* requires write access.

14464 CX [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 14465 argument.

14466 CX [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

14467 CX	[ENAMETOOLONG]	The length of the <i>filename</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
14468		
14469		
14470 CX	[ENFILE]	The maximum allowable number of files is currently open in the system.
14471 CX	[ENOENT]	A component of <i>filename</i> does not name an existing file or <i>filename</i> is an empty string.
14472		
14473 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.
14474		
14475 CX	[ENOTDIR]	A component of the path prefix is not a directory.
14476 CX	[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
14477		
14478 CX	[EOVERFLOW]	The named file is a regular file and the size of the file cannot be represented correctly in an object of type off_t .
14479		
14480 CX	[EROFS]	The named file resides on a read-only file system and <i>mode</i> requires write access.
14481		
14482	The <i>freopen()</i> function may fail if:	
14483 CX	[EBADF]	The mode with which the file descriptor underlying the stream was opened does not support the requested mode when <i>filename</i> is a null pointer.
14484		2
14485 CX	[EINVAL]	The value of the <i>mode</i> argument is not valid.
14486 CX	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path</i> argument.
14487		
14488 CX	[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
14489		
14490		
14491 CX	[ENOMEM]	Insufficient storage space is available.
14492 CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
14493		
14494 CX	[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and <i>mode</i> requires write access.
14495		

14496 EXAMPLES

14497 Directing Standard Output to a File

14498 The following example logs all standard output to the **/tmp/logfile** file.

```
14499 #include <stdio.h>
14500 ...
14501 FILE *fp;
14502 ...
14503 fp = freopen ("/tmp/logfile", "a+", stdout);
14504 ...
```

14505 **APPLICATION USAGE**

14506 The *freopen()* function is typically used to attach the preopened *streams* associated with *stdin*,
 14507 *stdout*, and *stderr* to other files.

14508 **RATIONALE**

14509 None.

14510 **FUTURE DIRECTIONS**

14511 None.

14512 **SEE ALSO**

14513 *fclose()*, *fopen()*, *fdopen()*, *mbsinit()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 14514 <*stdio.h*>

14515 **CHANGE HISTORY**

14516 First released in Issue 1. Derived from Issue 1 of the SVID.

14517 **Issue 5**

14518 The DESCRIPTION is updated to indicate that the orientation of the stream is cleared and the
 14519 conversion state of the stream is set to an initial conversion state by a successful call to the
 14520 *freopen()* function.

14521 Large File Summit extensions are added.

14522 **Issue 6**

14523 Extensions beyond the ISO C standard are marked.

14524 The following new requirements on POSIX implementations derive from alignment with the
 14525 Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file
 14527 description. This change is to support large files.

14528 • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support
 14529 large files.

- The [ELOOP] mandatory error condition is added.

- A second [ENAMETOOLONG] is added as an optional error condition.

- The [EINVAL], [ENOMEM], [ENXIO], and [ETXTBSY] optional error conditions are added.

14533 The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The *freopen()* prototype is updated.

- The DESCRIPTION is updated.

14536 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
 14537 [ELOOP] error condition is added.

14538 The DESCRIPTION is updated regarding failure to close, changing the “file” to “file descriptor”.

14539 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/40 is applied, adding the sentence “In this
 14540 case, the file descriptor associated with the stream need not be closed if the call to *freopen()* 2
 14541 succeeds.” to the DESCRIPTION. 2

14542 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/41 is applied, adding an mandatory 2
 14543 [EBADF] error, and an optional [EBADF] error to the ERRORS section. 2

14544 NAME

14545 *frexp*, *frexpf*, *frexpl* — extract mantissa and exponent from a double precision number

14546 SYNOPSIS

```
14547        #include <math.h>
14548        double frexp(double num, int *exp);
14549        float frexpf(float num, int *exp);
14550        long double frexpl(long double num, int *exp);
```

14551 DESCRIPTION

14552 CX The functionality described on this reference page is aligned with the ISO C standard. Any
14553 conflict between the requirements described here and the ISO C standard is unintentional. This
14554 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

14555 These functions shall break a floating-point number *num* into a normalized fraction and an
14556 integral power of 2. The integer exponent shall be stored in the **int** object pointed to by *exp*.

14557 RETURN VALUE

14558 For finite arguments, these functions shall return the value *x*, such that *x* has a magnitude in the
14559 interval [$\frac{1}{2}, 1$) or 0, and *num* equals *x* times 2 raised to the power **exp*.

14560 MX If *num* is NaN, a NaN shall be returned, and the value of **exp* is unspecified.

14561 If *num* is ± 0 , ± 0 shall be returned, and the value of **exp* shall be 0.

14562 If *num* is $\pm \infty$, *num* shall be returned, and the value of **exp* is unspecified.

14563 ERRORS

14564 No errors are defined.

14565 EXAMPLES

14566 None.

14567 APPLICATION USAGE

14568 None.

14569 RATIONALE

14570 None.

14571 FUTURE DIRECTIONS

14572 None.

14573 SEE ALSO

14574 *isnan()*, *ldexp()*, *modf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**math.h**>

14575 CHANGE HISTORY

14576 First released in Issue 1. Derived from Issue 1 of the SVID.

14577 Issue 5

14578 The DESCRIPTION is updated to indicate how an application should check for an error. This
14579 text was previously published in the APPLICATION USAGE section.

14580 Issue 6

14581 The *frexpf()* and *frexpl()* functions are added for alignment with the ISO/IEC 9899:1999
14582 standard.

- 14583 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
14584 revised to align with the ISO/IEC 9899:1999 standard.
- 14585 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
14586 marked.

14587 NAME

14588 fscanf, scanf, sscanf — convert formatted input

14589 SYNOPSIS

```
14590     #include <stdio.h>
14591
14592     int fscanf(FILE *restrict stream, const char *restrict format, ... );
14593     int scanf(const char *restrict format, ... );
14594     int sscanf(const char *restrict s, const char *restrict format, ... );
```

14594 DESCRIPTION

14595 CX 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.

14598 The *fscanf()* function shall read from the named input *stream*. The *scanf()* function shall read from the standard input stream *stdin*. The *sscanf()* function shall read from the string *s*. Each function reads bytes, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments shall be evaluated but otherwise ignored.

14605 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion specifier character % (see below) is replaced by the sequence "%*n\$*", where *n* is a decimal integer in the range [1,{NL_ARGMAX}]. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages. In format strings containing the "%*n\$*" form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format string more than once.

14612 The *format* can contain either form of a conversion specification—that is, % or "%*n\$*"—but the two forms cannot be mixed within a single *format* string. The only exception to this is that %% or %* can be mixed with the "%*n\$*" form. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*-1)th, are pointers.

14617 CX The *fscanf()* function in all its forms shall allow detection of a language-dependent radix character in the input string. The radix character is defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.') .

14621 The format is a character string, beginning and ending in its initial shift state, if any, composed of zero or more directives. Each directive is composed of one of the following: one or more white-space characters (<space>s, <tab>s, <newline>s, <vertical-tab>s, or <form-feed>s); an ordinary character (neither '%' nor a white-space character); or a conversion specification. Each conversion specification is introduced by the character '%' or the character sequence "%*n\$*", after which the following appear in sequence:

- 14627 • An optional assignment-suppressing character '*' .
- 14628 • An optional non-zero decimal integer that specifies the maximum field width.
- 14629 • An option length modifier that specifies the size of the receiving object.
- 14630 • A *conversion specifier* character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

14632 The *fscanf()* functions shall execute each directive of the format in turn. If a directive fails, as
14633 detailed below, the function shall return. Failures are described as input failures (due to the
14634 unavailability of input bytes) or matching failures (due to inappropriate input).

14635 A directive composed of one or more white-space characters shall be executed by reading input
14636 until no more valid input can be read, or up to the first byte which is not a white-space character,
14637 which remains unread.

14638 A directive that is an ordinary character shall be executed as follows: the next byte shall be read
14639 from the input and compared with the byte that comprises the directive; if the comparison
14640 shows that they are not equivalent, the directive shall fail, and the differing and subsequent
14641 bytes shall remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a
14642 character from being read, the directive shall fail.

14643 A directive that is a conversion specification defines a set of matching input sequences, as
14644 described below for each conversion character. A conversion specification shall be executed in
14645 the following steps.

14646 Input white-space characters (as specified by *isspace()*) shall be skipped, unless the conversion
14647 specification includes a *[*, *c*, *C*, or *n* conversion specifier.

14648 An item shall be read from the input, unless the conversion specification includes an *n*
14649 conversion specifier. An input item shall be defined as the longest sequence of input bytes (up to
14650 any specified maximum field width, which may be measured in characters or bytes dependent
14651 on the conversion specifier) which is an initial subsequence of a matching sequence. The first
14652 byte, if any, after the input item shall remain unread. If the length of the input item is 0, the
14653 execution of the conversion specification shall fail; this condition is a matching failure, unless
14654 end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is
14655 an input failure.

14656 Except in the case of a *%* conversion specifier, the input item (or, in the case of a *%n* conversion
14657 specification, the count of input bytes) shall be converted to a type appropriate to the conversion
14658 character. If the input item is not a matching sequence, the execution of the conversion
14659 specification fails; this condition is a matching failure. Unless assignment suppression was
14660 indicated by a *'*'*, the result of the conversion shall be placed in the object pointed to by the
14661 first argument following the *format* argument that has not already received a conversion result if
14662 XSI
14663 the conversion specification is introduced by *%*, or in the *n* argument if introduced by the
14664 character sequence *"%n\$"*. If this object does not have an appropriate type, or if the result of the
conversion cannot be represented in the space provided, the behavior is undefined.

14665 The length modifiers and their meanings are:

14666 *hh* Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
14667 argument with type pointer to **signed char** or **unsigned char**.

14668 *h* Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
14669 argument with type pointer to **short** or **unsigned short**.

14670 *l* (ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
14671 argument with type pointer to **long** or **unsigned long**; that a following *a*, *A*, *e*, *E*, *f*, *F*, *g*,
14672 or *G* conversion specifier applies to an argument with type pointer to **double**; or that a
14673 following *c*, *s*, or *[* conversion specifier applies to an argument with type pointer to
14674 **wchar_t**.

14675 *ll* (ell-ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
14676 argument with type pointer to **long long** or **unsigned long long**.
14677

- 14678 j Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
14679 argument with type pointer to **intmax_t** or **uintmax_t**.
- 14680 z Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
14681 argument with type pointer to **size_t** or the corresponding signed integer type.
- 14682 t Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
14683 argument with type pointer to **ptrdiff_t** or the corresponding **unsigned** type.
- 14684 L Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an
14685 argument with type pointer to **long double**.
- 14686 If a length modifier appears with any conversion specifier other than as specified above, the
14687 behavior is undefined.
- 14688 The following conversion specifiers are valid:
- 14689 d Matches an optionally signed decimal integer, whose format is the same as expected for
14690 the subject sequence of *strtol()* with the value 10 for the *base* argument. In the absence
14691 of a size modifier, the application shall ensure that the corresponding argument is a
14692 pointer to **int**.
- 14693 i Matches an optionally signed integer, whose format is the same as expected for the
14694 subject sequence of *strtol()* with 0 for the *base* argument. In the absence of a size
14695 modifier, the application shall ensure that the corresponding argument is a pointer to
14696 **int**.
- 14697 o Matches an optionally signed octal integer, whose format is the same as expected for
14698 the subject sequence of *strtoul()* with the value 8 for the *base* argument. In the absence
14699 of a size modifier, the application shall ensure that the corresponding argument is a
14700 pointer to **unsigned**.
- 14701 u Matches an optionally signed decimal integer, whose format is the same as expected for
14702 the subject sequence of *strtoul()* with the value 10 for the *base* argument. In the absence
14703 of a size modifier, the application shall ensure that the corresponding argument is a
14704 pointer to **unsigned**.
- 14705 x Matches an optionally signed hexadecimal integer, whose format is the same as
14706 expected for the subject sequence of *strtoul()* with the value 16 for the *base* argument. In
14707 the absence of a size modifier, the application shall ensure that the corresponding
14708 argument is a pointer to **unsigned**.
- 14709 a, e, f, g Matches an optionally signed floating-point number, infinity, or NaN, whose format is
14710 the same as expected for the subject sequence of *strtod()*. In the absence of a size
14711 modifier, the application shall ensure that the corresponding argument is a pointer to
14712 **float**.
- 14713 If the *fprintf()* family of functions generates character string representations for infinity
14714 and NaN (a symbolic entity encoded in floating-point format) to support
14715 IEEE Std 754-1985, the *fscanf()* family of functions shall recognize them as input.
- 14716 s Matches a sequence of bytes that are not white-space characters. The application shall
14717 ensure that the corresponding argument is a pointer to the initial byte of an array of
14718 **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a
14719 terminating null character code, which shall be added automatically.
- 14720 If an l (ell) qualifier is present, the input is a sequence of characters that begins in the
14721 initial shift state. Each character shall be converted to a wide character as if by a call to

- 14723 the *mbrtowc()* function, with the conversion state described by an **mbstate_t** object
14724 initialized to zero before the first character is converted. The application shall ensure
14725 that the corresponding argument is a pointer to an array of **wchar_t** large enough to
14726 accept the sequence and the terminating null wide character, which shall be added
14727 automatically.
- 14728 [Matches a non-empty sequence of bytes from a set of expected bytes (the *scanset*). The
14729 normal skip over white-space characters shall be suppressed in this case. The
14730 application shall ensure that the corresponding argument is a pointer to the initial byte
14731 of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence
14732 and a terminating null byte, which shall be added automatically.
- 14733 l If an *l* (ell) qualifier is present, the input is a sequence of characters that begins in the
14734 initial shift state. Each character in the sequence shall be converted to a wide character
14735 as if by a call to the *mbrtowc()* function, with the conversion state described by an
14736 **mbstate_t** object initialized to zero before the first character is converted. The
14737 application shall ensure that the corresponding argument is a pointer to an array of
14738 **wchar_t** large enough to accept the sequence and the terminating null wide character,
14739 which shall be added automatically.
- 14740 [The conversion specification includes all subsequent bytes in the *format* string up to
14741 and including the matching right square bracket (']'). The bytes between the square
14742 brackets (the *scanlist*) comprise the scanset, unless the byte after the left square bracket
14743 is a circumflex (^), in which case the scanset contains all bytes that do not appear in
14744 the scanlist between the circumflex and the right square bracket. If the conversion
14745 specification begins with "[]" or "[^]", the right square bracket is included in the
14746 scanlist and the next right square bracket is the matching right square bracket that ends
14747 the conversion specification; otherwise, the first right square bracket is the one that
14748 ends the conversion specification. If a '-' is in the scanlist and is not the first character,
14749 nor the second where the first character is a '^', nor the last character, the behavior is
14750 implementation-defined.
- 14751 c Matches a sequence of bytes of the number specified by the field width (1 if no field
14752 width is present in the conversion specification). The application shall ensure that the
14753 corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**,
14754 or **unsigned char** large enough to accept the sequence. No null byte is added. The
14755 normal skip over white-space characters shall be suppressed in this case.
- 14756 [If an *l* (ell) qualifier is present, the input shall be a sequence of characters that begins in the
14757 initial shift state. Each character in the sequence is converted to a wide character as
14758 if by a call to the *mbrtowc()* function, with the conversion state described by an
14759 **mbstate_t** object initialized to zero before the first character is converted. The
14760 application shall ensure that the corresponding argument is a pointer to an array of
14761 **wchar_t** large enough to accept the resulting sequence of wide characters. No null wide
14762 character is added.
- 14763 p Matches an implementation-defined set of sequences, which shall be the same as the set
14764 of sequences that is produced by the %p conversion specification of the corresponding
14765 *fprintf()* functions. The application shall ensure that the corresponding argument is a
14766 pointer to a pointer to **void**. The interpretation of the input item is implementation-
14767 defined. If the input item is a value converted earlier during the same program
14768 execution, the pointer that results shall compare equal to that value; otherwise, the
14769 behavior of the %p conversion specification is undefined.
- 14770 n No input is consumed. The application shall ensure that the corresponding argument is
14771 a pointer to the integer into which shall be written the number of bytes read from the

14772 input so far by this call to the *fscanf()* functions. Execution of a %n conversion
14773 specification shall not increment the assignment count returned at the completion of
14774 execution of the function. No argument shall be converted, but one shall be consumed.
14775 If the conversion specification includes an assignment-suppressing character or a field
14776 width, the behavior is undefined.

14777 XSI C Equivalent to 1c.

14778 XSI S Equivalent to 1s.

14779 % Matches a single '%' character; no conversion or assignment occurs. The complete
14780 conversion specification shall be %%.

14781 If a conversion specification is invalid, the behavior is undefined.

14782 The conversion specifiers A, E, F, G, and X are also valid and shall be equivalent to a, e, f, g, and
14783 x, respectively.

14784 If end-of-file is encountered during input, conversion shall be terminated. If end-of-file occurs
14785 before any bytes matching the current conversion specification (except for %n) have been read
14786 (other than leading white-space characters, where permitted), execution of the current
14787 conversion specification shall terminate with an input failure. Otherwise, unless execution of the
14788 current conversion specification is terminated with a matching failure, execution of the
14789 following conversion specification (if any) shall be terminated with an input failure.

14790 Reaching the end of the string in *sscanf()* shall be equivalent to encountering end-of-file for
14791 *fscanf()*.

14792 If conversion terminates on a conflicting input, the offending input is left unread in the input.
14793 Any trailing white space (including <newline>s) shall be left unread unless matched by a
14794 conversion specification. The success of literal matches and suppressed assignments is only
14795 directly determinable via the %n conversion specification.

14796 CX The *fscanf()* and *scanf()* functions may mark the *st_atime* field of the file associated with *stream*
14797 for update. The *st_atime* field shall be marked for update by the first successful execution of
14798 *fgetc()*, *fgets()*, *fread()*, *getc()*, *getchar()*, *gets()*, *fscanf()*, or *fscanf()* using *stream* that returns data
14799 not supplied by a prior call to *ungetc()*.

14800 RETURN VALUE

14801 Upon successful completion, these functions shall return the number of successfully matched
14802 and assigned input items; this number can be zero in the event of an early matching failure. If
14803 the input ends before the first matching failure or conversion, EOF shall be returned. If a read
14804 CX error occurs, the error indicator for the stream is set, EOF shall be returned, and *errno* shall be set
14805 to indicate the error.

14806 ERRORS

14807 For the conditions under which the *fscanf()* functions fail and may fail, refer to *fgetc()* or
14808 *fgetwc()*.

14809 In addition, *fscanf()* may fail if:

14810 XSI [EILSEQ] Input byte sequence does not form a valid character.

14811 XSI [EINVAL] There are insufficient arguments.

14812 EXAMPLES

14813 The call:

```
14814       int i, n; float x; char name[50];  
14815       n = scanf("%d%f%s", &i, &x, name);
```

14816 with the input line:

```
14817       25 54.32E-1 Hamster
```

14818 assigns to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* contains the string "Hamster".

14820 The call:

```
14821       int i; float x; char name[50];  
14822       (void) scanf("%2d%f*d %[0123456789]", &i, &x, name);
```

14823 with input:

```
14824       56789 0123 56a72
```

14825 assigns 56 to *i*, 789.0 to *x*, skips 0123, and places the string "56\0" in *name*. The next call to *getchar()* shall return the character 'a'.

14827 Reading Data into an Array

14828 The following call uses *fscanf()* to read three floating-point numbers from standard input into the *input* array.

```
14830       float input[3]; fscanf (stdin, "%f %f %f", input, input+1, input+2);
```

14831 APPLICATION USAGE

14832 If the application calling *fscanf()* has any objects of type **wint_t** or **wchar_t**, it must also include the **<wchar.h>** header to have these objects defined.

14834 RATIONALE

14835 This function is aligned with the ISO/IEC 9899:1999 standard, and in doing so a few "obvious"
14836 things were not included. Specifically, the set of characters allowed in a scanset is limited to
14837 single-byte characters. In other similar places, multi-byte characters have been permitted, but
14838 for alignment with the ISO/IEC 9899:1999 standard, it has not been done here. Applications
14839 needing this could use the corresponding wide-character functions to achieve the desired
14840 results.

14841 FUTURE DIRECTIONS

14842 None.

14843 SEE ALSO

14844 *getc()*, *printf()*, *setlocale()*, *strtod()*, *strtol()*, *strtoul()*, *wcrtomb()*, the Base Definitions volume of
14845 IEEE Std 1003.1-2001, Chapter 7, Locale, **<langinfo.h>**, **<stdio.h>**, **<wchar.h>**

14846 CHANGE HISTORY

14847 First released in Issue 1. Derived from Issue 1 of the SVID.

14848 Issue 5

14849 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the **l** (ell) qualifier is
14850 now defined for the **c**, **s**, and **[** conversion specifiers.

14851 The DESCRIPTION is updated to indicate that if infinity and NaN can be generated by the
14852 *fprintf()* family of functions, then they are recognized by the *fscanf()* family.

14853 **Issue 6**

14854 The Open Group Corrigenda U021/7 and U028/10 are applied. These correct several
14855 occurrences of “characters” in the text which have been replaced with the term “bytes”.

14856 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

14857 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 14858 • The prototypes for *fscanf()*, *scanf()*, and *sscanf()* are updated.
- 14859 • The DESCRIPTION is updated.
- 14860 • The hh, ll, j, t, and z length modifiers are added.
- 14861 • The a, A, and F conversion characters are added.

14862 The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion
14863 specification” consistently.

14864 NAME

14865 fseek, fseeko — reposition a file-position indicator in a stream

14866 SYNOPSIS

```
14867       #include <stdio.h>
14868       int fseek(FILE *stream, long offset, int whence);
14869 CX       int fseeko(FILE *stream, off_t offset, int whence);
```

14871 DESCRIPTION

14872 CX 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.

14875 The *fseek()* function shall set the file-position indicator for the stream pointed to by *stream*. If a
14876 read or write error occurs, the error indicator for the stream shall be set and *fseek()* fails.

14877 The new position, measured in bytes from the beginning of the file, shall be obtained by adding
14878 *offset* to the position specified by *whence*. The specified point is the beginning of the file for
14879 SEEK_SET, the current value of the file-position indicator for SEEK_CUR, or end-of-file for
14880 SEEK_END.

14881 If the stream is to be used with wide-character input/output functions, the application shall
14882 ensure that *offset* is either 0 or a value returned by an earlier call to *ftell()* on the same stream and
14883 *whence* is SEEK_SET.

14884 A successful call to *fseek()* shall clear the end-of-file indicator for the stream and undo any effects
14885 of *ungetc()* and *ungetwc()* on the same stream. After an *fseek()* call, the next operation on an
14886 update stream may be either input or output.

14887 CX If the most recent operation, other than *ftell()*, on a given stream is *fflush()*, the file offset in the
14888 underlying open file description shall be adjusted to reflect the location specified by *fseek()*.

14889 The *fseek()* function shall allow the file-position indicator to be set beyond the end of existing
14890 data in the file. If data is later written at this point, subsequent reads of data in the gap shall
14891 return bytes with the value 0 until data is actually written into the gap.

14892 The behavior of *fseek()* on devices which are incapable of seeking is implementation-defined.
14893 The value of the file offset associated with such a device is undefined.

14894 If the stream is writable and buffered data had not been written to the underlying file, *fseek()*
14895 shall cause the unwritten data to be written to the file and shall mark the *st_ctime* and *st_mtime*
14896 fields of the file for update.

14897 In a locale with state-dependent encoding, whether *fseek()* restores the stream's shift state is
14898 implementation-defined.

14899 The *fseeko()* function shall be equivalent to the *fseek()* function except that the *offset* argument is
14900 of type **off_t**.

14901 RETURN VALUE

14902 CX The *fseek()* and *fseeko()* functions shall return 0 if they succeed.

14903 CX Otherwise, they shall return -1 and set *errno* to indicate the error.

14904 ERRORS

14905 CX The *fseek()* and *fseeko()* functions shall fail if, either the *stream* is unbuffered or the *stream*'s
14906 buffer needed to be flushed, and the call to *fseek()* or *fseeko()* causes an underlying *lseek()* or
14907 *write()* to be invoked, and:

14908 CX	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor and the thread would be delayed in the write operation.	2
14909			
14910 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.	
14911			
14912 CX	[EFBIG]	An attempt was made to write a file that exceeds the maximum file size.	
14913 XSI	[EFBIG]	An attempt was made to write a file that exceeds the process' file size limit.	
14914 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.	
14915			
14916 CX	[EINTR]	The write operation was terminated due to the receipt of a signal, and no data was transferred.	
14917			
14918 CX	[EINVAL]	The whence argument is invalid. The resulting file-position indicator would be set to a negative value.	
14919			
14920 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.	
14921			
14922			
14923			
14924			
14925 CX	[ENOSPC]	There was no free space remaining on the device containing the file.	
14926 CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.	
14927			
14928 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 .	
14929			
14930 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 .	
14931			
14932 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.	
14933			
14934 CX	[ESPIPE]	The file descriptor underlying <i>stream</i> is associated with a pipe or FIFO.	

14935 EXAMPLES

14936 None.

14937 APPLICATION USAGE

14938 None.

14939 RATIONALE

14940 None.

14941 FUTURE DIRECTIONS

14942 None.

14943 SEE ALSO14944 *fopen()*, *fsetpos()*, *ftell()*, *getrlimit()*, *lseek()*, *rewind()*, *ulimit()*, *ungetc()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**stdio.h**>**14946 CHANGE HISTORY**

14947 First released in Issue 1. Derived from Issue 1 of the SVID.

14948 Issue 5

14949 Normative text previously in the APPLICATION USAGE section is moved to the
14950 DESCRIPTION.

14951 Large File Summit extensions are added.

14952 Issue 6

14953 Extensions beyond the ISO C standard are marked.

14954 The following new requirements on POSIX implementations derive from alignment with the
14955 Single UNIX Specification:

- The *fseeko()* function is added.

- The [EFBIG], [EOVERFLOW], and [ENXIO] mandatory error conditions are added.

14958 The following change is incorporated for alignment with the FIPS requirements:

- The [EINTR] error is no longer an indication that the implementation does not report partial
transfers.

14961 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

14962 The DESCRIPTION is updated to explicitly state that *fseek()* sets the file-position indicator, and
14963 then on error the error indicator is set and *fseek()* fails. This is for alignment with the
14964 ISO/IEC 9899:1999 standard.

14965 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/42 is applied, updating the [EAGAIN] 2
14966 error in the ERRORS section from “the process would be delayed” to “the thread would be 2
14967 delayed”. 2

14968 NAME

14969 fsetpos — set current file position

14970 SYNOPSIS

```
14971 #include <stdio.h>
14972 int fsetpos(FILE *stream, const fpos_t *pos);
```

14973 DESCRIPTION

14974 CX 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.

14977 The *fsetpos()* function shall set the file position and state indicators for the stream pointed to by *stream* according to the value of the object pointed to by *pos*, which the application shall ensure is a value obtained from an earlier call to *fgetpos()* on the same stream. If a read or write error occurs, the error indicator for the stream shall be set and *fsetpos()* fails.

14981 A successful call to the *fsetpos()* function shall clear the end-of-file indicator for the stream and undo any effects of *ungetc()* on the same stream. After an *fsetpos()* call, the next operation on an update stream may be either input or output.

14984 CX The behavior of *fsetpos()* on devices which are incapable of seeking is implementation-defined. The value of the file offset associated with such a device is undefined.

14986 RETURN VALUE

14987 The *fsetpos()* function shall return 0 if it succeeds; otherwise, it shall return a non-zero value and set *errno* to indicate the error.

14989 ERRORS

14990 CX The *fsetpos()* function shall fail if, either the *stream* is unbuffered or the *stream*'s buffer needed to be flushed, and the call to *fsetpos()* causes an underlying *lseek()* or *write()* to be invoked, and:

14992 CX [EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the thread would be delayed in the write operation. 2

14994 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.

14996 CX [EFBIG] An attempt was made to write a file that exceeds the maximum file size.

14997 XSI [EFBIG] An attempt was made to write a file that exceeds the process' file size limit.

14998 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.

15000 CX [EINTR] The write operation was terminated due to the receipt of a signal, and no data was transferred. 1

15002 CX [EIO] A physical I/O error has occurred, or the process is a member of a background process group attempting to perform a *write()* 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.

15007 CX [ENOSPC] There was no free space remaining on the device containing the file.

15008 CX [ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

15010 CX	[EPIPE]	The file descriptor underlying <i>stream</i> is associated with a pipe or FIFO.
15011 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.
15012		

15013 EXAMPLES

15014 None.

15015 APPLICATION USAGE

15016 None.

15017 RATIONALE

15018 None.

15019 FUTURE DIRECTIONS

15020 None.

15021 SEE ALSO

15022 *open()*, *ftell()*, *lseek()*, *rewind()*, *ungetc()*, *write()*, the Base Definitions volume of
 15023 IEEE Std 1003.1-2001, <stdio.h>

15024 CHANGE HISTORY

15025 First released in Issue 4. Derived from the ISO C standard.

15026 Issue 6

15027 Extensions beyond the ISO C standard are marked.

15028 An additional [ESPIPE] error condition is added for sockets.

15029 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

15030 The DESCRIPTION is updated to clarify that the error indicator is set for the stream on a read or
 15031 write error. This is for alignment with the ISO/IEC 9899:1999 standard.

15032 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/21 is applied, deleting an erroneous 1
 15033 [EINVAL] error case from the ERRORS section. 1

15034 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/43 is applied, updating the [EAGAIN] 2
 15035 error in the ERRORS section from “the process would be delayed” to “the thread would be 2
 15036 delayed”. 2

15037 NAME

15038 fstat — get file status

15039 SYNOPSIS

15040 #include <sys/stat.h>

15041 int fstat(int *fildes*, struct stat **buf*);

15042 DESCRIPTION

15043 The *fstat()* function shall obtain information about an open file associated with the file descriptor *fildes*, and shall write it to the area pointed to by *buf*.

15045 SHM If *fildes* references a shared memory object, the implementation shall update in the **stat** structure pointed to by the *buf* argument only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid. The implementation may update other fields and flags.

15049 TYM If *fildes* references a typed memory object, the implementation shall update in the **stat** structure pointed to by the *buf* argument only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid. The implementation may update other fields and flags.

15053 The *buf* argument is a pointer to a **stat** structure, as defined in <sys/stat.h>, into which information is placed concerning the file.

15055 The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime* shall have meaningful values for all other file types defined in this volume of IEEE Std 1003.1-2001. The value of the member *st_nlink* shall be set to the number of links to the file.

15059 An implementation that provides additional or alternative file access control mechanisms may, under implementation-defined conditions, cause *fstat()* to fail.

15061 The *fstat()* function shall update any time-related fields as described in the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.7, File Times Update, before writing into the **stat** structure.

15064 RETURN VALUE

15065 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

15067 ERRORS

15068 The *fstat()* function shall fail if:

15069 [EBADF] The *fildes* argument is not a valid file descriptor.

15070 [EIO] An I/O error occurred while reading from the file system.

15071 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

15074 The *fstat()* function may fail if:

15075 [EOVERFLOW] One of the values is too large to store into the structure pointed to by the *buf* argument.

15077 EXAMPLES

15078 Obtaining File Status Information

15079 The following example shows how to obtain file status information for a file named
15080 `/home/cnd/mod1`. The structure variable `buffer` is defined for the `stat` structure. The
15081 `/home/cnd/mod1` file is opened with read/write privileges and is passed to the open file
15082 descriptor `fildes`.

```
15083 #include <sys/types.h>
15084 #include <sys/stat.h>
15085 #include <fcntl.h>

15086 struct stat buffer;
15087 int status;
15088 ...
15089 fildes = open( "/home/cnd/mod1" , O_RDWR );
15090 status = fstat(fildes, &buffer);
```

15091 APPLICATION USAGE

15092 None.

15093 RATIONALE

15094 None.

15095 FUTURE DIRECTIONS

15096 None.

15097 SEE ALSO

15098 `Istat()`, `stat()`, the Base Definitions volume of IEEE Std 1003.1-2001, <`sys/stat.h`>, <`sys/types.h`>

15099 CHANGE HISTORY

15100 First released in Issue 1. Derived from Issue 1 of the SVID.

15101 Issue 5

15102 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

15103 Large File Summit extensions are added.

15104 Issue 6

15105 In the SYNOPSIS, the optional include of the <`sys/types.h`> header is removed.

15106 The following new requirements on POSIX implementations derive from alignment with the
15107 Single UNIX Specification:

- 15108 • The requirement to include <`sys/types.h`> has been removed. Although <`sys/types.h`> was
15109 required for conforming implementations of previous POSIX specifications, it was not
15110 required for UNIX applications.
- 15111 • The [EIO] mandatory error condition is added.
- 15112 • The [EOVERFLOW] mandatory error condition is added. This change is to support large
15113 files.
- 15114 • The [EOVERFLOW] optional error condition is added.

15115 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
15116 shared memory object semantics apply to typed memory objects.

15117 NAME

15118 fstatvfs, statvfs — get file system information

15119 SYNOPSIS

```
15120 XSI #include <sys/statvfs.h>
15121     int fstatvfs(int fildes, struct statvfs *buf);
15122     int statvfs(const char *restrict path, struct statvfs *restrict buf);
15123
```

15124 DESCRIPTION

15125 The *fstatvfs()* function shall obtain information about the file system containing the file
 15126 referenced by *fildes*.

15127 The *statvfs()* function shall obtain information about the file system containing the file named by
 15128 *path*.

15129 For both functions, the *buf* argument is a pointer to a **statvfs** structure that shall be filled. Read,
 15130 write, or execute permission of the named file is not required.

15131 The following flags can be returned in the *f_flag* member:

15132 ST_RDONLY Read-only file system.

15133 ST_NOSUID Setuid/setgid bits ignored by *exec*.

15134 It is unspecified whether all members of the **statvfs** structure have meaningful values on all file
 15135 systems.

15136 RETURN VALUE

15137 Upon successful completion, *statvfs()* shall return 0. Otherwise, it shall return -1 and set *errno* to
 15138 indicate the error.

15139 ERRORS

15140 The *fstatvfs()* and *statvfs()* functions shall fail if:

15141 [EIO] An I/O error occurred while reading the file system.

15142 [EINTR] A signal was caught during execution of the function.

15143 [EOVERFLOW] One of the values to be returned cannot be represented correctly in the
 15144 structure pointed to by *buf*.

15145 The *fstatvfs()* function shall fail if:

15146 [EBADF] The *fildes* argument is not an open file descriptor.

15147 The *statvfs()* function shall fail if:

15148 [EACCES] Search permission is denied on a component of the path prefix.

15149 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 15150 argument.

15151 [ENAMETOOLONG]

15152 The length of a pathname exceeds {PATH_MAX} or a pathname component is
 15153 longer than {NAME_MAX}.

15154 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

15155 [ENOTDIR] A component of the path prefix of *path* is not a directory.

15156 The *statvfs()* function may fail if:

15157 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
15158 resolution of the *path* argument.

15159 [ENAMETOOLONG]
15160 Pathname resolution of a symbolic link produced an intermediate result
15161 whose length exceeds {PATH_MAX}.

15162 EXAMPLES

15163 Obtaining File System Information Using fstatvfs()

15164 The following example shows how to obtain file system information for the file system upon
15165 which the file named **/home/cnd/mod1** resides, using the *fstatvfs()* function. The
15166 **/home/cnd/mod1** file is opened with read/write privileges and the open file descriptor is passed
15167 to the *fstatvfs()* function.

```
15168 #include <statvfs.h>
15169 #include <fcntl.h>
15170
15171 struct statvfs buffer;
15172 int status;
15173 ...
15174 fildes = open( "/home/cnd/mod1" , O_RDWR );
15175 status = fstatvfs(fildes, &buffer);
```

15175 Obtaining File System Information Using statvfs()

15176 The following example shows how to obtain file system information for the file system upon
15177 which the file named **/home/cnd/mod1** resides, using the *statvfs()* function.

```
15178 #include <statvfs.h>
15179
15180 struct statvfs buffer;
15181 int status;
15182 ...
15183 status = statvfs( "/home/cnd/mod1" , &buffer );
```

15183 APPLICATION USAGE

15184 None.

15185 RATIONALE

15186 None.

15187 FUTURE DIRECTIONS

15188 None.

15189 SEE ALSO

15190 *chmod()*, *chown()*, *creat()*, *dup()*, *exec*, *fcntl()*, *link()*, *mknod()*, *open()*, *pipe()*, *read()*, *time()*,
15191 *unlink()*, *utime()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/statvfs.h>

15192 CHANGE HISTORY

15193 First released in Issue 4, Version 2.

15194 Issue 5

15195 Moved from X/OPEN UNIX extension to BASE.

15196 Large File Summit extensions are added.

15197 Issue 6

- 15198 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
- 15199 The **restrict** keyword is added to the *statvfs()* prototype for alignment with the
15200 ISO/IEC 9899:1999 standard.
- 15201 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
15202 [ELOOP] error condition is added.

15203 **NAME**

15204 *fsync* — synchronize changes to a file

15205 **SYNOPSIS**

15206 FSC

```
#include <unistd.h>
```

15207

```
int fsync(int fildes);
```

15208

15209 **DESCRIPTION**

15210 The *fsync()* function shall request that all data for the open file descriptor named by *fildes* is to be transferred to the storage device associated with the file described by *fildes*. The nature of the transfer is implementation-defined. The *fsync()* function shall not return until the system has completed that action or until an error is detected.

2
2

15214 SIO If *_POSIX_SYNCHRONIZED_IO* is defined, the *fsync()* function shall force all currently queued I/O operations associated with the file indicated by file descriptor *fildes* to the synchronized I/O completion state. All I/O operations shall be completed as defined for synchronized I/O file integrity completion.

15218 **RETURN VALUE**

15219 Upon successful completion, *fsync()* shall return 0. Otherwise, -1 shall be returned and *errno* set to indicate the error. If the *fsync()* function fails, outstanding I/O operations are not guaranteed to have been completed.

15222 **ERRORS**

15223 The *fsync()* function shall fail if:

15224 [EBADF] The *fildes* argument is not a valid descriptor.

15225 [EINTR] The *fsync()* function was interrupted by a signal.

15226 [EINVAL] The *fildes* argument does not refer to a file on which this operation is possible.

15227 [EIO] An I/O error occurred while reading from or writing to the file system.

15228 In the event that any of the queued I/O operations fail, *fsync()* shall return the error conditions defined for *read()* and *write()*.

15230 **EXAMPLES**

15231 None.

15232 **APPLICATION USAGE**

15233 The *fsync()* function should be used by programs which require modifications to a file to be completed before continuing; for example, a program which contains a simple transaction facility might use it to ensure that all modifications to a file or files caused by a transaction are recorded.

15237 **RATIONALE**

15238 The *fsync()* function is intended to force a physical write of data from the buffer cache, and to assure that after a system crash or other failure that all data up to the time of the *fsync()* call is recorded on the disk. Since the concepts of "buffer cache", "system crash", "physical write", and "non-volatile storage" are not defined here, the wording has to be more abstract.

15242 If *_POSIX_SYNCHRONIZED_IO* is not defined, the wording relies heavily on the conformance document to tell the user what can be expected from the system. It is explicitly intended that a null implementation is permitted. This could be valid in the case where the system cannot assure non-volatile storage under any circumstances or when the system is highly fault-tolerant and the functionality is not required. In the middle ground between these extremes, *fsync()* might or might not actually cause data to be written where it is safe from a power failure. The

15248 conformance document should identify at least that one configuration exists (and how to obtain
15249 that configuration) where this can be assured for at least some files that the user can select to use
15250 for critical data. It is not intended that an exhaustive list is required, but rather sufficient
15251 information is provided so that if critical data needs to be saved, the user can determine how the
15252 system is to be configured to allow the data to be written to non-volatile storage.

15253 It is reasonable to assert that the key aspects of *fsync()* are unreasonable to test in a test suite.
15254 That does not make the function any less valuable, just more difficult to test. A formal
15255 conformance test should probably force a system crash (power shutdown) during the test for
15256 this condition, but it needs to be done in such a way that automated testing does not require this
15257 to be done except when a formal record of the results is being made. It would also not be
15258 unreasonable to omit testing for *fsync()*, allowing it to be treated as a quality-of-implementation
15259 issue.

15260 FUTURE DIRECTIONS

15261 None.

15262 SEE ALSO

15263 *sync()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

15264 CHANGE HISTORY

15265 First released in Issue 3.

15266 Issue 5

15267 Aligned with *fsync()* in the POSIX Realtime Extension. Specifically, the DESCRIPTION and
15268 RETURN VALUE sections are much expanded, and the ERRORS section is updated to indicate
15269 that *fsync()* can return the error conditions defined for *read()* and *write()*.

15270 Issue 6

15271 This function is marked as part of the File Synchronization option.

15272 The following new requirements on POSIX implementations derive from alignment with the
15273 Single UNIX Specification:

- 15274 • The [EINVAL] and [EIO] mandatory error conditions are added.

15275 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/44 is applied, applying an editorial 2
15276 rewording of the DESCRIPTION. No change in meaning is intended. 2

15277 NAME

15278 *ftell*, *ftello* — return a file offset in a stream

15279 SYNOPSIS

15280 #include <stdio.h>

15281 long *ftell*(FILE **stream*);
15282 CX off_t *ftello*(FILE **stream*);
15283

15284 DESCRIPTION

15285 CX 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.

15288 The *ftell*() function shall obtain the current value of the file-position indicator for the stream pointed to by *stream*.

15290 CX The *ftello*() function shall be equivalent to *ftell*(), except that the return value is of type off_t.

15291 RETURN VALUE

15292 CX Upon successful completion, *ftell*() and *ftello*() shall return the current value of the file-position indicator for the stream measured in bytes from the beginning of the file.

15294 CX Otherwise, *ftell*() and *ftello*() shall return -1, cast to long and off_t respectively, and set *errno* to indicate the error.

15296 ERRORS

15297 CX The *ftell*() and *ftello*() functions shall fail if:

15298 CX [EBADF] The file descriptor underlying *stream* is not an open file descriptor.

15299 CX [EOVERFLOW] For *ftell*(), the current file offset cannot be represented correctly in an object of
15300 type long.

15301 CX [EOVERFLOW] For *ftello*(), the current file offset cannot be represented correctly in an object
15302 of type off_t.

15303 CX [ESPIPE] The file descriptor underlying *stream* is associated with a pipe or FIFO.

15304 The *ftell*() function may fail if:

15305 CX [ESPIPE] The file descriptor underlying *stream* is associated with a socket.

15306 EXAMPLES

15307 None.

15308 APPLICATION USAGE

15309 None.

15310 RATIONALE

15311 None.

15312 FUTURE DIRECTIONS

15313 None.

15314 SEE ALSO

15315 *fgetpos*(), *fopen*(), *fseek*(), *lseek*(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

15316 CHANGE HISTORY

15317 First released in Issue 1. Derived from Issue 1 of the SVID.

15318 Issue 5

15319 Large File Summit extensions are added.

15320 Issue 6

15321 Extensions beyond the ISO C standard are marked.

15322 The following new requirements on POSIX implementations derive from alignment with the
15323 Single UNIX Specification:

- 15324 • The *f_{tello}()* function is added.
- 15325 • The [EOVERFLOW] error conditions are added.

15326 An additional [ESPIPE] error condition is added for sockets.

15327 NAME

15328 *ftime* — get date and time (**LEGACY**)

15329 SYNOPSIS

15330 XSI `#include <sys/timeb.h>`

15331 `int ftime(struct timeb *tp);`

15332

15333 DESCRIPTION

15334 The *ftime()* function shall set the *time* and *millitm* members of the **timeb** structure pointed to by *tp* to contain the seconds and milliseconds portions, respectively, of the current time in seconds since the Epoch. The contents of the *timezone* and *dstflag* members of *tp* after a call to *ftime()* are unspecified.

15338 The system clock need not have millisecond granularity. Depending on any granularity (particularly a granularity of one) renders code non-portable.

15340 RETURN VALUE

15341 Upon successful completion, the *ftime()* function shall return 0; otherwise, -1 shall be returned.

15342 ERRORS

15343 No errors are defined.

15344 EXAMPLES**15345 Getting the Current Time and Date**

15346 The following example shows how to get the current system time values using the *ftime()* function. The **timeb** structure pointed to by *tp* is filled with the current system time values for *time* and *millitm*.

```
15349        #include <sys/timeb.h>
15350
15351        struct timeb tp;
15352        int            status;
15353
15354        ...          
15355        status = ftime(&tp);
```

15354 APPLICATION USAGE

15355 For applications portability, the *time()* function should be used to determine the current time instead of *ftime()*. Realtime applications should use *clock_gettime()* to determine the current time instead of *ftime()*.

15358 RATIONALE

15359 None.

15360 FUTURE DIRECTIONS

15361 This function may be withdrawn in a future version.

15362 SEE ALSO

15363 *clock_getres()*, *ctime()*, *gettimeofday()*, *time()*, the Base Definitions volume of
15364 IEEE Std 1003.1-2001, <sys/timeb.h>

15365 CHANGE HISTORY

15366 First released in Issue 4, Version 2.

15367 Issue 5

15368 Moved from X/OPEN UNIX extension to BASE.

15369 Normative text previously in the APPLICATION USAGE section is moved to the
15370 DESCRIPTION.

15371 Issue 6

15372 This function is marked LEGACY.

15373 The DESCRIPTION is updated to refer to “seconds since the Epoch” rather than “seconds since
15374 00:00:00 UTC (Coordinated Universal Time), January 1 1970” for consistency with other *time*
15375 functions.

15376 NAME

15377 ftok — generate an IPC key

15378 SYNOPSIS

15379 XSI #include <sys/ipc.h>

15380 key_t ftok(const char *path, int id);

15381

15382 DESCRIPTION

15383 The *ftok()* function shall return a key based on *path* and *id* that is usable in subsequent calls to
15384 *msgget()*, *semget()*, and *shmget()*. The application shall ensure that the *path* argument is the
15385 pathname of an existing file that the process is able to *stat()*.

15386 The *ftok()* function shall return the same key value for all paths that name the same file, when
15387 called with the same *id* value, and return different key values when called with different *id*
15388 values or with paths that name different files existing on the same file system at the same time. It
15389 is unspecified whether *ftok()* shall return the same key value when called again after the file
15390 named by *path* is removed and recreated with the same name.

15391 Only the low-order 8-bits of *id* are significant. The behavior of *ftok()* is unspecified if these bits
15392 are 0.

15393 RETURN VALUE

15394 Upon successful completion, *ftok()* shall return a key. Otherwise, *ftok()* shall return (key_t)-1
15395 and set *errno* to indicate the error.

15396 ERRORS

15397 The *ftok()* function shall fail if:

15398 [EACCES] Search permission is denied for a component of the path prefix.

15399 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
15400 argument.

15401 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
15402 component is longer than {NAME_MAX}.

15404 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

15405 [ENOTDIR] A component of the path prefix is not a directory.

15406 The *ftok()* function may fail if:

15407 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
15408 resolution of the *path* argument.

15409 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
15410 whose length exceeds {PATH_MAX}.

15412 EXAMPLES

15413 Getting an IPC Key

15414 The following example gets a unique key that can be used by the IPC functions *semget()*,
15415 *msgget()*, and *shmget()*. The key returned by *ftok()* for this example is based on the ID value *S*
15416 and the pathname **/tmp**.

```
15417 #include <sys/IPC.h>
15418 ...
15419 key_t key;
15420 char *path = "/tmp";
15421 int id = 'S';
15422 key = ftok(path, id);
```

15423 Saving an IPC Key

15424 The following example gets a unique key based on the pathname **/tmp** and the ID value *a*. It
15425 also assigns the value of the resulting key to the *semkey* variable so that it will be available to a
15426 later call to *semget()*, *msgget()*, or *shmget()*.

```
15427 #include <sys/IPC.h>
15428 ...
15429 key_t semkey;
15430 if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
15431     perror("IPC error: ftok");
15432     exit(1);
}
```

15433 APPLICATION USAGE

15434 For maximum portability, *id* should be a single-byte character.

15435 RATIONALE

15436 None.

15437 FUTURE DIRECTIONS

15438 None.

15439 SEE ALSO

15440 *msgget()*, *semget()*, *shmget()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/IPC.h>

15441 CHANGE HISTORY

15442 First released in Issue 4, Version 2.

15443 Issue 5

15444 Moved from X/OPEN UNIX extension to BASE.

15445 Issue 6

15446 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

15447 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
15448 [ELOOP] error condition is added.

15449 NAME

15450 ftruncate — truncate a file to a specified length

15451 SYNOPSIS

```
15452 #include <unistd.h>
15453 int ftruncate(int fildes, off_t length);
```

15454 DESCRIPTION

15455 If *fildes* is not a valid file descriptor open for writing, the *ftruncate()* function shall fail.

15456 If *fildes* refers to a regular file, the *ftruncate()* function shall cause the size of the file to be
 15457 truncated to *length*. If the size of the file previously exceeded *length*, the extra data shall no
 15458 longer be available to reads on the file. If the file previously was smaller than this size,
 15459 XSI *ftruncate()* shall either increase the size of the file or fail. XSI-conformant systems shall increase
 15460 the size of the file. If the file size is increased, the extended area shall appear as if it were zero-
 15461 filled. The value of the seek pointer shall not be modified by a call to *ftruncate()*.

15462 Upon successful completion, if *fildes* refers to a regular file, the *ftruncate()* function shall mark
 15463 for update the *st_ctime* and *st_mtime* fields of the file and the S_ISUID and S_ISGID bits of the file
 15464 mode may be cleared. If the *ftruncate()* function is unsuccessful, the file is unaffected.

15465 XSI If the request would cause the file size to exceed the soft file size limit for the process, the
 15466 request shall fail and the implementation shall generate the SIGXFSZ signal for the thread.

15467 If *fildes* refers to a directory, *ftruncate()* shall fail.

15468 If *fildes* refers to any other file type, except a shared memory object, the result is unspecified.

15469 SHM If *fildes* refers to a shared memory object, *ftruncate()* shall set the size of the shared memory
 15470 object to *length*.

15471 MF|SHM If the effect of *ftruncate()* is to decrease the size of a shared memory object or memory mapped
 15472 file and whole pages beyond the new end were previously mapped, then the whole pages
 15473 beyond the new end shall be discarded.

15474 MPR If the Memory Protection option is supported, references to discarded pages shall result in the
 15475 generation of a SIGBUS signal; otherwise, the result of such references is undefined.

15476 MF|SHM If the effect of *ftruncate()* is to increase the size of a shared memory object, it is unspecified
 15477 whether the contents of any mapped pages between the old end-of-file and the new are flushed
 15478 to the underlying object.

15479 RETURN VALUE

15480 Upon successful completion, *ftruncate()* shall return 0; otherwise, -1 shall be returned and *errno*
 15481 set to indicate the error.

15482 ERRORS

15483 The *ftruncate()* function shall fail if:

15484 [EINTR] A signal was caught during execution.

15485 [EINVAL] The *length* argument was less than 0.

15486 [EFBIG] or [EINVAL]
 15487 The *length* argument was greater than the maximum file size.

15488 XSI [EFBIG] The file is a regular file and *length* is greater than the offset maximum
 15489 established in the open file description associated with *fildes*.

15490 [EIO] An I/O error occurred while reading from or writing to a file system.

15491	[EBADF] or [EINVAL]	
15492		The <i>fd</i> argument is not a file descriptor open for writing.
15493	[EINVAL]	The <i>fd</i> argument references a file that was opened without write permission.
15494		
15495	[EROFS]	The named file resides on a read-only file system.

15496 EXAMPLES

15497 None.

15498 APPLICATION USAGE

15499 None.

15500 RATIONALE

15501 The *ftruncate()* function is part of IEEE Std 1003.1-2001 as it was deemed to be more useful than
15502 *truncate()*. The *truncate()* function is provided as an XSI extension.

15503 FUTURE DIRECTIONS

15504 None.

15505 SEE ALSO

15506 *open()*, *truncate()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

15507 CHANGE HISTORY

15508 First released in Issue 4, Version 2.

15509 Issue 5

15510 Moved from X/OPEN UNIX extension to BASE and aligned with *ftruncate()* in the POSIX
15511 Realtime Extension. Specifically, the DESCRIPTION is extensively reworded and [EROFS] is
15512 added to the list of mandatory errors that can be returned by *ftruncate()*.

15513 Large File Summit extensions are added.

15514 Issue 6

15515 The *truncate()* function is split out into a separate reference page.

15516 The following new requirements on POSIX implementations derive from alignment with the
15517 Single UNIX Specification:

- The DESCRIPTION is changed to indicate that if the file size is changed, and if the file is a regular file, the S_ISUID and S_ISGID bits in the file mode may be cleared.

15518 The following changes were made to align with the IEEE P1003.1a draft standard:
15519

- The DESCRIPTION text is updated.

15520 XSI-conformant systems are required to increase the size of the file if the file was previously
15521 smaller than the size requested.

15524 NAME

15525 ftrylockfile — stdio locking functions

15526 SYNOPSIS

15527 TSF #include <stdio.h>

15528 int ftrylockfile(FILE *file);

15529

15530 DESCRIPTION

15531 Refer to *flockfile()*.

15532 NAME

15533 ftw — traverse (walk) a file tree

15534 SYNOPSIS

15535 XSI #include <ftw.h>

```
15536 int ftw(const char *path, int (*fn)(const char *,
15537         const struct stat *ptr, int flag), int ndirs);
```

15538

15539 DESCRIPTION

15540 The *ftw()* function shall recursively descend the directory hierarchy rooted in *path*. For each
15541 object in the hierarchy, *ftw()* shall call the function pointed to by *fn*, passing it a pointer to a
15542 null-terminated character string containing the name of the object, a pointer to a **stat** structure
15543 containing information about the object, and an integer. Possible values of the integer, defined
15544 in the <ftw.h> header, are:

15545 FTW_D For a directory.

15546 FTW_DNR For a directory that cannot be read.

15547 FTW_F For a file.

15548 FTW_SL For a symbolic link (but see also FTW_NS below).

15549 FTW_NS For an object other than a symbolic link on which *stat()* could not successfully be
15550 executed. If the object is a symbolic link and *stat()* failed, it is unspecified whether
15551 *ftw()* passes FTW_SL or FTW_NS to the user-supplied function.

15552 If the integer is FTW_DNR, descendants of that directory shall not be processed. If the integer is
15553 FTW_NS, the **stat** structure contains undefined values. An example of an object that would
15554 cause FTW_NS to be passed to the function pointed to by *fn* would be a file in a directory with
15555 read but without execute (search) permission.

15556 The *ftw()* function shall visit a directory before visiting any of its descendants.

15557 The *ftw()* function shall use at most one file descriptor for each level in the tree.

15558 The argument *ndirs* should be in the range [1,{OPEN_MAX}].

15559 The tree traversal shall continue until either the tree is exhausted, an invocation of *fn* returns a
15560 non-zero value, or some error, other than [EACCES], is detected within *ftw()*.

15561 The *ndirs* argument shall specify the maximum number of directory streams or file descriptors
15562 or both available for use by *ftw()* while traversing the tree. When *ftw()* returns it shall close any
15563 directory streams and file descriptors it uses not counting any opened by the application-
15564 supplied *fn* function.

15565 The results are unspecified if the application-supplied *fn* function does not preserve the current
15566 working directory.

15567 The *ftw()* function need not be reentrant. A function that is not required to be reentrant is not
15568 required to be thread-safe.

15569 RETURN VALUE

15570 If the tree is exhausted, *ftw()* shall return 0. If the function pointed to by *fn* returns a non-zero
15571 value, *ftw()* shall stop its tree traversal and return whatever value was returned by the function
15572 pointed to by *fn*. If *ftw()* detects an error, it shall return -1 and set *errno* to indicate the error.

15573 If *ftw()* encounters an error other than [EACCES] (see FTW_DNR and FTW_NS above), it shall
15574 return -1 and set *errno* to indicate the error. The external variable *errno* may contain any error

15575 value that is possible when a directory is opened or when one of the *stat* functions is executed on
15576 a directory or file.

15577 ERRORS

15578 The *ftw()* function shall fail if:

15579 [EACCES] Search permission is denied for any component of *path* or read permission is
15580 denied for *path*.

15581 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
15582 argument.

15583 [ENAMETOOLONG]

15584 The length of the *path* argument exceeds {PATH_MAX} or a pathname
15585 component is longer than {NAME_MAX}.

15586 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

15587 [ENOTDIR] A component of *path* is not a directory.

15588 [EOVERFLOW] A field in the *stat* structure cannot be represented correctly in the current
15589 programming environment for one or more files found in the file hierarchy.

15590 The *ftw()* function may fail if:

15591 [EINVAL] The value of the *ndirs* argument is invalid.

15592 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
15593 resolution of the *path* argument.

15594 [ENAMETOOLONG]

15595 Pathname resolution of a symbolic link produced an intermediate result
15596 whose length exceeds {PATH_MAX}.

15597 In addition, if the function pointed to by *fn* encounters system errors, *errno* may be set
15598 accordingly.

15599 EXAMPLES

15600 Walking a Directory Structure

15601 The following example walks the current directory structure, calling the *fn* function for every
15602 directory entry, using at most 10 file descriptors:

```
15603 #include <ftw.h>
15604 ...
15605 if (ftw(".", fn, 10) != 0) {
15606     perror("ftw");
15607 }
```

15608 APPLICATION USAGE

15609 The *ftw()* function may allocate dynamic storage during its operation. If *ftw()* is forcibly
15610 terminated, such as by *longjmp()* or *siglongjmp()* being executed by the function pointed to by *fn*
15611 or an interrupt routine, *ftw()* does not have a chance to free that storage, so it remains
15612 permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has
15613 occurred, and arrange to have the function pointed to by *fn* return a non-zero value at its next
15614 invocation.

15615 RATIONALE

15616 None.

15617 FUTURE DIRECTIONS

15618 None.

15619 SEE ALSO

15620 *longjmp()*, *Istat()*, *malloc()*, *nftw()*, *opendir()*, *siglongjmp()*, *stat()*, the Base Definitions volume of
15621 IEEE Std 1003.1-2001, <ftw.h>, <sys/stat.h>

15622 CHANGE HISTORY

15623 First released in Issue 1. Derived from Issue 1 of the SVID.

15624 Issue 5

15625 UX codings in the DESCRIPTION, RETURN VALUE, and ERRORS sections are changed to EX.

15626 Issue 6

15627 The ERRORS section is updated as follows:

- 15628 • The wording of the mandatory [ELOOP] error condition is updated.
- 15629 • A second optional [ELOOP] error condition is added.
- 15630 • The [EOVERFLOW] mandatory error condition is added.

15631 Text is added to the DESCRIPTION to say that the *ftw()* function need not be reentrant and that
15632 the results are unspecified if the application-supplied *fn* function does not preserve the current
15633 working directory.

15634 **NAME**15635 **funlockfile** — stdio locking functions15636 **SYNOPSIS**

15637 TSF #include <stdio.h>

15638 void funlockfile(FILE *file);

15639

15640 **DESCRIPTION**15641 Refer to *flockfile()*.

15642 NAME

15643 fwide — set stream orientation

15644 SYNOPSIS

```
15645 #include <stdio.h>
15646 #include <wchar.h>
15647 int fwide(FILE *stream, int mode);
```

15648 DESCRIPTION

15649 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15650 conflict between the requirements described here and the ISO C standard is unintentional. This
15651 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

15652 The *fwide()* function shall determine the orientation of the stream pointed to by *stream*. If *mode* is
15653 greater than zero, the function first attempts to make the stream wide-oriented. If *mode* is less
15654 than zero, the function first attempts to make the stream byte-oriented. Otherwise, *mode* is zero
15655 and the function does not alter the orientation of the stream.

15656 If the orientation of the stream has already been determined, *fwide()* shall not change it.

15657 CX Since no return value is reserved to indicate an error, an application wishing to check for error
15658 situations should set *errno* to 0, then call *fwide()*, then check *errno*, and if it is non-zero, assume
15659 an error has occurred.

15660 RETURN VALUE

15661 The *fwide()* function shall return a value greater than zero if, after the call, the stream has wide-
15662 orientation, a value less than zero if the stream has byte-orientation, or zero if the stream has no
15663 orientation.

15664 ERRORS

15665 The *fwide()* function may fail if:

15666 CX [EBADF] The *stream* argument is not a valid stream.

15667 EXAMPLES

15668 None.

15669 APPLICATION USAGE

15670 A call to *fwide()* with *mode* set to zero can be used to determine the current orientation of a
15671 stream.

15672 RATIONALE

15673 None.

15674 FUTURE DIRECTIONS

15675 None.

15676 SEE ALSO

15677 The Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

15678 CHANGE HISTORY

15679 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
15680 (E).

15681 Issue 6

15682 Extensions beyond the ISO C standard are marked.

15683 NAME

15684 fwprintf, swprintf, wprintf — print formatted wide-character output

15685 SYNOPSIS

```
15686 #include <stdio.h>
15687 #include <wchar.h>
15688 int fwprintf(FILE *restrict stream, const wchar_t *restrict format, ...);
15689 int swprintf(wchar_t *restrict ws, size_t n,
15690               const wchar_t *restrict format, ...);
15691 int wprintf(const wchar_t *restrict format, ...);
```

15692 DESCRIPTION

15693 CX 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.

15694 The *fwprintf()* function shall place output on the named output *stream*. The *wprintf()* function shall place output on the standard output stream *stdout*. The *swprintf()* function shall place output followed by the null wide character in consecutive wide characters starting at **ws*; no more than *n* wide characters shall be written, including a terminating null wide character, which is always added (unless *n* is zero).

15695 Each of these functions shall convert, format, and print its arguments under control of the *format* wide-character string. The *format* is composed of zero or more directives: *ordinary wide-characters*, which are simply copied to the output stream, and *conversion specifications*, each of which results in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

15696 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion specifier wide character % (see below) is replaced by the sequence "%n\$ ", where *n* is a decimal integer in the range [1,{NL_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of *format* wide-character strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

15697 The *format* can contain either numbered argument specifications (that is, "%n\$" and "*m\$"), or unnumbered argument conversion specifications (that is, % and *), but not both. The only exception to this is that %% can be mixed with the "%n\$ " form. The results of mixing numbered and unnumbered argument specifications in a *format* wide-character string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*-1)th, are specified in the *format* wide-character string.

15698 In *format* wide-character strings containing the "%n\$ " form of conversion specification, numbered arguments in the argument list can be referenced from the *format* wide-character string as many times as required.

15699 In *format* wide-character strings containing the % form of conversion specification, each argument in the argument list shall be used exactly once.

15700 CX All forms of the *fwprintf()* function allow for the insertion of a locale-dependent radix character in the output string, output as a wide-character value. The radix character is defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.') .

15701 XSI Each conversion specification is introduced by the '%' wide character or by the wide-character sequence "%n\$ ", after which the following appear in sequence:

- 15730 • Zero or more *flags* (in any order), which modify the meaning of the conversion specification.
- 15731 • An optional minimum *field width*. If the converted value has fewer wide characters than the field width, it shall be padded with spaces by default on the left; it shall be padded on the right, if the left-adjustment flag ('-'), described below, is given to the field width. The field width takes the form of an asterisk ('*'), described below, or a decimal integer.
- 15735 • An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversion specifiers; the number of digits to appear after the radix character for the a, A, e, E, f, and F conversion specifiers; the maximum number of significant digits for the g and G conversion specifiers; or the maximum number of wide characters to be printed from a string in the s conversion specifiers. The precision takes the form of a period ('.') followed either by an asterisk ('*'), described below, or an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with any other conversion wide character, the behavior is undefined.
- 15743 • An optional length modifier that specifies the size of the argument.
- 15744 • A *conversion specifier* wide character that indicates the type of conversion to be applied.

15745 A field width, or precision, or both, may be indicated by an asterisk ('*'). In this case an
 15746 argument of type **int** supplies the field width or precision. Applications shall ensure that
 15747 arguments specifying field width, or precision, or both appear in that order before the argument,
 15748 if any, to be converted. A negative field width is taken as a '-' flag followed by a positive field
 15749 **XSI** width. A negative precision is taken as if the precision were omitted. In *format* wide-character
 15750 strings containing the "%n\$" form of a conversion specification, a field width or precision may
 15751 be indicated by the sequence "*m\$ ", where m is a decimal integer in the range
 15752 [1,{NL_ARGMAX}] giving the position in the argument list (after the *format* argument) of an
 15753 integer argument containing the field width or precision, for example:

```
wprintf(L"%1$d:%2$.3$.*3$d:%4$.3$d\n", hour, min, precision, sec);
```

15755 The flag wide characters and their meanings are:

- 15756 **XSI** ' The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %F, %g, or %G)
 15757 shall be formatted with thousands' grouping wide characters. For other conversions,
 15758 the behavior is undefined. The numeric grouping wide character is used.
- 15759 - The result of the conversion shall be left-justified within the field. The conversion shall
 15760 be right-justified if this flag is not specified.
- 15761 + The result of a signed conversion shall always begin with a sign ('+' or '-'). The
 15762 conversion shall begin with a sign only when a negative value is converted if this flag is
 15763 not specified.
- 15764 <space> If the first wide character of a signed conversion is not a sign, or if a signed conversion
 15765 results in no wide characters, a <space> shall be prefixed to the result. This means that
 15766 if the <space> and '+' flags both appear, the <space> flag shall be ignored.
- 15767 # Specifies that the value is to be converted to an alternative form. For o conversion, it
 15768 increases the precision (if necessary) to force the first digit of the result to be 0. For x or
 15769 X conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For a, A, e,
 15770 E, f, F, g, and G conversion specifiers, the result shall always contain a radix character,
 15771 even if no digits follow it. Without this flag, a radix character appears in the result of
 15772 these conversions only if a digit follows it. For g and G conversion specifiers, trailing
 15773 zeros shall *not* be removed from the result as they normally are. For other conversion
 15774 specifiers, the behavior is undefined.

15775 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.

15782 The length modifiers and their meanings are:

15783 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.

15788 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.

15793 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.

15798 ll (ell-ell) 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.

15802 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.

15805 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.

15808 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.

15811 L Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a **long double** argument.

15813 If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

15815 The conversion specifiers and their meanings are:

15816 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.

- 15821 o The **unsigned** argument shall be converted to unsigned octal format in the style "ddddd". 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.
- 15826 u The **unsigned** argument shall be converted to unsigned decimal format in the style "ddddd". 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.
- 15831 x The **unsigned** argument shall be converted to unsigned hexadecimal format in the style "ddddd"; 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.
- 15836 X Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead of "abcdef".
- 15838 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.
- 15844 A 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(*n-char-sequence*)"; which style, and the meaning of any *n-char-sequence*, is implementation-defined. The F conversion specifier produces "INF", "INFINITY", or "NAN" instead of "inf", "infinity", or "nan", respectively.
- 15850 e, E The **double** argument shall be converted in the style "[-]d.ddde \pm 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.
- 15858 A A **double** argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.
- 15860 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.
- 15867 A A **double** argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

15869	a, A	A double argument representing a floating-point number shall be converted in the style "[-]0x.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.
15883		A double argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.
15885	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.
15888	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>mbtowc()</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.
15897		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.
15903	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.
15906	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.
15911 XSI	C	Equivalent to lc.
15912 XSI	S	Equivalent to ls.
15913	%	Output a '%' wide character; no argument shall be converted. The entire conversion specification shall be %%.
15914		

15915 If a conversion specification does not match one of the above forms, the behavior is undefined.

15916 In no case does a nonexistent or small field width cause truncation of a field; if the result of a
15917 conversion is wider than the field width, the field shall be expanded to contain the conversion
15918 result. Characters generated by *fwprintf()* and *wprintf()* shall be printed as if *fputwc()* had been
15919 called.

15920 For a and A conversions, if FLT_RADIX is not a power of 2 and the result is not exactly
15921 representable in the given precision, the result should be one of the two adjacent numbers in
15922 hexadecimal floating style with the given precision, with the extra stipulation that the error
15923 should have a correct sign for the current rounding direction.

15924 For e, E, f, F, g, and G conversion specifiers, if the number of significant decimal digits is at most
15925 DECIMAL_DIG, then the result should be correctly rounded. If the number of significant
15926 decimal digits is more than DECIMAL_DIG but the source value is exactly representable with
15927 DECIMAL_DIG digits, then the result should be an exact representation with trailing zeros.
15928 Otherwise, the source value is bounded by two adjacent decimal strings L < U, both having
15929 DECIMAL_DIG significant digits; the value of the resultant decimal string D should satisfy L <= D <= U,
15930 with the extra stipulation that the error should have a correct sign for the current
15931 rounding direction.

15932 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the call to a
15933 successful execution of *fwprintf()* or *wprintf()* and the next successful completion of a call to
15934 *fflush()* or *fclose()* on the same stream, or a call to *exit()* or *abort()*.

15935 RETURN VALUE

15936 Upon successful completion, these functions shall return the number of wide characters
15937 transmitted, excluding the terminating null wide character in the case of *swprintf()*, or a negative
15938 CX value if an output error was encountered, and set *errno* to indicate the error.

15939 If *n* or more wide characters were requested to be written, *swprintf()* shall return a negative
15940 CX value, and set *errno* to indicate the error.

15941 ERRORS

15942 For the conditions under which *fwprintf()* and *wprintf()* fail and may fail, refer to *fputwc()*.

15943 In addition, all forms of *fwprintf()* may fail if:

15944 XSI [EILSEQ] A wide-character code that does not correspond to a valid character has been
15945 detected.

15946 XSI [EINVAL] There are insufficient arguments.

15947 In addition, *wprintf()* and *fwprintf()* may fail if:

15948 XSI [ENOMEM] Insufficient storage space is available.

15949 EXAMPLES

15950 To print the language-independent date and time format, the following statement could be used:

15951 *wprintf(format, weekday, month, day, hour, min);*

15952 For American usage, *format* could be a pointer to the wide-character string:

15953 L "%s, %s %d, %d:%.2d\n"

15954 producing the message:

15955 Sunday, July 3, 10:02

15956 whereas for German usage, *format* could be a pointer to the wide-character string:

15957 L"%1\$s, %3\$d. %2\$s, %4\$d:%5\$.2d\n"

15958 producing the message:

15959 Sonntag, 3. Juli, 10:02

15960 APPLICATION USAGE

15961 None.

15962 RATIONALE

15963 None.

15964 FUTURE DIRECTIONS

15965 None.

15966 SEE ALSO

15967 *btowc()*, *fputwc()*, *fwscanf()*, *mbrtowc()*, *setlocale()*, the Base Definitions volume of
15968 IEEE Std 1003.1-2001, Chapter 7, Locale, <stdio.h>, <wchar.h>

15969 CHANGE HISTORY

15970 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
15971 (E).

15972 Issue 6

15973 The Open Group Corrigendum U040/1 is applied to the RETURN VALUE section, describing
15974 the case if *n* or more wide characters are requested to be written using *swprintf()*.

15975 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

15976 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 15977 • The prototypes for *fwprintf()*, *swprintf()*, and *wprintf()* are updated.

- 15978 • The DESCRIPTION is updated.

- 15979 • The hh, ll, j, t, and z length modifiers are added.

- 15980 • The a, A, and F conversion characters are added.

- 15981 • XSI shading is removed from the description of character string representations of infinity
15982 and NaN floating-point values.

15983 The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion
15984 specification” consistently.

15985 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

15986 **NAME**

15987 `fwrite` — binary output

15988 **SYNOPSIS**

```
15989        #include <stdio.h>
15990
15991        size_t fwrite(const void *restrict ptr, size_t size, size_t nitems,
15991              FILE *restrict stream);
```

15992 **DESCRIPTION**

15993 CX The functionality described on this reference page is aligned with the ISO C standard. Any
15994 conflict between the requirements described here and the ISO C standard is unintentional. This
15995 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

15996 The `fwrite()` function shall write, from the array pointed to by `ptr`, up to `nitems` elements whose
15997 size is specified by `size`, to the stream pointed to by `stream`. For each object, `size` calls shall be
15998 made to the `fputc()` function, taking the values (in order) from an array of **unsigned char** exactly
15999 overlaying the object. The file-position indicator for the stream (if defined) shall be advanced by
16000 the number of bytes successfully written. If an error occurs, the resulting value of the file-
16001 position indicator for the stream is unspecified.

16002 CX The `st_ctime` and `st_mtime` fields of the file shall be marked for update between the successful
16003 execution of `fwrite()` and the next successful completion of a call to `fflush()` or `fclose()` on the
16004 same stream, or a call to `exit()` or `abort()`.

16005 **RETURN VALUE**

16006 The `fwrite()` function shall return the number of elements successfully written, which may be
16007 less than `nitems` if a write error is encountered. If `size` or `nitems` is 0, `fwrite()` shall return 0 and the
16008 state of the stream remains unchanged. Otherwise, if a write error occurs, the error indicator for
16009 CX the stream shall be set, and `errno` shall be set to indicate the error.

16010 **ERRORS**

16011 Refer to `fputc()`.

16012 **EXAMPLES**

16013 None.

16014 **APPLICATION USAGE**

16015 Because of possible differences in element length and byte ordering, files written using `fwrite()`
16016 are application-dependent, and possibly cannot be read using `fread()` by a different application
16017 or by the same application on a different processor.

16018 **RATIONALE**

16019 None.

16020 **FUTURE DIRECTIONS**

16021 None.

16022 **SEE ALSO**

16023 `ferror()`, `fopen()`, `printf()`, `putc()`, `puts()`, `write()`, the Base Definitions volume of
16024 IEEE Std 1003.1-2001, `<stdio.h>`

16025 **CHANGE HISTORY**

16026 First released in Issue 1. Derived from Issue 1 of the SVID.

16027 **Issue 6**

16028 Extensions beyond the ISO C standard are marked.

16029 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

16030

- The *fwrite()* prototype is updated.
- The DESCRIPTION is updated to clarify how the data is written out using *fputc()*.

16031

16032 NAME

16033 fwscanf, swscanf, wscanf — convert formatted wide-character input

16034 SYNOPSIS

```

16035 #include <stdio.h>
16036 #include <wchar.h>
16037 int fwscanf(FILE *restrict stream, const wchar_t *restrict format, ... );
16038 int swscanf(const wchar_t *restrict ws,
16039     const wchar_t *restrict format, ... );
16040 int wscanf(const wchar_t *restrict format, ... );

```

16041 DESCRIPTION

16042 CX 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.

16043 The *fwscanf()* function shall read from the named input *stream*. The *wscanf()* function shall read from the standard input stream *stdin*. The *swscanf()* function shall read from the wide-character string *ws*. Each function reads wide characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control wide-character string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the *format* is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

16044 XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion specifier wide character % (see below) is replaced by the sequence "%n\$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}]. This feature provides for the definition of *format* wide-character strings that select arguments in an order appropriate to specific languages. In *format* wide-character strings containing the "%n\$" form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the *format* wide-character string more than once.

16045 The *format* can contain either form of a conversion specification—that is, % or "%n\$"—but the two forms cannot normally be mixed within a single *format* wide-character string. The only exception to this is that %% or %* can be mixed with the "%n\$" form. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*-1)th, are pointers.

16046 CX The *fwscanf()* function in all its forms allows for detection of a language-dependent radix character in the input string, encoded as a wide-character value. The radix character is defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.').

16047 The *format* is a wide-character string composed of zero or more directives. Each directive is composed of one of the following: one or more white-space wide characters (<space>s, <tab>s, <newline>s, <vertical-tab>s, or <form-feed>s); an ordinary wide character (neither '%' nor a white-space character); or a conversion specification. Each conversion specification is introduced by a '%' or the sequence "%n\$" after which the following appear in sequence:

- 16048 • An optional assignment-suppressing character '*'.
- 16049 • An optional non-zero decimal integer that specifies the maximum field width.
- 16050 • An optional length modifier that specifies the size of the receiving object.

- 16077 • A conversion specifier wide character that specifies the type of conversion to be applied. The
16078 valid conversion specifiers are described below.

16079 The *fwscanf()* functions shall execute each directive of the format in turn. If a directive fails, as
16080 detailed below, the function shall return. Failures are described as input failures (due to the
16081 unavailability of input bytes) or matching failures (due to inappropriate input).

16082 A directive composed of one or more white-space wide characters is executed by reading input
16083 until no more valid input can be read, or up to the first wide character which is not a white-
16084 space wide character, which remains unread.

16085 A directive that is an ordinary wide character shall be executed as follows. The next wide
16086 character is read from the input and compared with the wide character that comprises the
16087 directive; if the comparison shows that they are not equivalent, the directive shall fail, and the
16088 differing and subsequent wide characters remain unread. Similarly, if end-of-file, an encoding
16089 error, or a read error prevents a wide character from being read, the directive shall fail.

16090 A directive that is a conversion specification defines a set of matching input sequences, as
16091 described below for each conversion wide character. A conversion specification is executed in
16092 the following steps.

16093 Input white-space wide characters (as specified by *iswspace()*) shall be skipped, unless the
16094 conversion specification includes a *[*, *c*, or *n* conversion specifier.

16095 An item shall be read from the input, unless the conversion specification includes an *n*
16096 conversion specifier wide character. An input item is defined as the longest sequence of input
16097 wide characters, not exceeding any specified field width, which is an initial subsequence of a
16098 matching sequence. The first wide character, if any, after the input item shall remain unread. If
16099 the length of the input item is zero, the execution of the conversion specification shall fail; this
16100 condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented
16101 input from the stream, in which case it is an input failure.

16102 Except in the case of a *%* conversion specifier, the input item (or, in the case of a *%n* conversion
16103 specification, the count of input wide characters) shall be converted to a type appropriate to the
16104 conversion wide character. If the input item is not a matching sequence, the execution of the
16105 conversion specification shall fail; this condition is a matching failure. Unless assignment
16106 suppression was indicated by a *'*'*, the result of the conversion shall be placed in the object
16107 pointed to by the first argument following the *format* argument that has not already received a
16108 XSI conversion result if the conversion specification is introduced by *%*, or in the *n*th argument if
16109 introduced by the wide-character sequence *"%n\$"*. If this object does not have an appropriate
16110 type, or if the result of the conversion cannot be represented in the space provided, the behavior
16111 is undefined.

16112 The length modifiers and their meanings are:

- 16113 hh Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
16114 argument with type pointer to **signed char** or **unsigned char**.
- 16115 h Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
16116 argument with type pointer to **short** or **unsigned short**.
- 16117 l (ell) Specifies that a following *d*, *i*, *o*, *u*, *x*, *X*, or *n* conversion specifier applies to an
16118 argument with type pointer to **long** or **unsigned long**; that a following *a*, *A*, *e*, *E*, *f*, *F*, *g*,
16119 or *G* conversion specifier applies to an argument with type pointer to **double**; or that a
16120 following *c*, *s*, or *[* conversion specifier applies to an argument with type pointer to
16121 **wchar_t**.

16122	l1 (ell-ell)	Specifies that a following d, i, o, u, x, or n conversion specifier applies to an argument with type pointer to long long or unsigned long long .
16123	j	Specifies that a following d, i, o, u, x, or n conversion specifier applies to an argument with type pointer to intmax_t or uintmax_t .
16124	z	Specifies that a following d, i, o, u, x, or n conversion specifier applies to an argument with type pointer to size_t or the corresponding signed integer type.
16125	t	Specifies that a following d, i, o, u, x, or n conversion specifier applies to an argument with type pointer to ptrdiff_t or the corresponding unsigned type.
16126	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 .
16127		If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.
16128		The following conversion specifier wide characters are valid:
16129	d	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of <i>wcstol()</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 .
16130	i	Matches an optionally signed integer, whose format is the same as expected for the subject sequence of <i>wcstol()</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 .
16131	o	Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of <i>wcstoul()</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 .
16132	u	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of <i>wcstoul()</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 .
16133	x	Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of <i>wcstoul()</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 .
16134	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>wcstod()</i> . In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to float .
16135		If the <i>fwprintf()</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>fwscanf()</i> family of functions shall recognize them as input.
16136	s	Matches a sequence of non white-space wide characters. If no l (ell) qualifier is present, characters from the input field shall be converted as if by repeated calls to the <i>wcrtnmb()</i> function, with the conversion state described by an mbstate_t object

- 16167 initialized to zero before the first wide character is converted. The application shall
16168 ensure that the corresponding argument is a pointer to a character array large enough
16169 to accept the sequence and the terminating null character, which shall be added
16170 automatically.
- 16171 Otherwise, the application shall ensure that the corresponding argument is a pointer to
16172 an array of **wchar_t** large enough to accept the sequence and the terminating null wide
16173 character, which shall be added automatically.
- 16174 [Matches a non-empty sequence of wide characters from a set of expected wide
16175 characters (the *scanset*). If no **l** (ell) qualifier is present, wide characters from the input
16176 field shall be converted as if by repeated calls to the **wcrtomb()** function, with the
16177 conversion state described by an **mbstate_t** object initialized to zero before the first
16178 wide character is converted. The application shall ensure that the corresponding
16179 argument is a pointer to a character array large enough to accept the sequence and the
16180 terminating null character, which shall be added automatically.
- 16181 If an **l** (ell) qualifier is present, the application shall ensure that the corresponding
16182 argument is a pointer to an array of **wchar_t** large enough to accept the sequence and
16183 the terminating null wide character, which shall be added automatically.
- 16184 The conversion specification includes all subsequent wide characters in the *format*
16185 string up to and including the matching right square bracket ('] '). The wide
16186 characters between the square brackets (the *scanlist*) comprise the scanset, unless the
16187 wide character after the left square bracket is a circumflex (' ^ '), in which case the
16188 scanset contains all wide characters that do not appear in the scanlist between the
16189 circumflex and the right square bracket. If the conversion specification begins with
16190 "[] " or "[^] ", the right square bracket is included in the scanlist and the next right
16191 square bracket is the matching right square bracket that ends the conversion
16192 specification; otherwise, the first right square bracket is the one that ends the
16193 conversion specification. If a '-' is in the scanlist and is not the first wide character,
16194 nor the second where the first wide character is a '^', nor the last wide character, the
16195 behavior is implementation-defined.
- 16196 c Matches a sequence of wide characters of exactly the number specified by the field
16197 width (1 if no field width is present in the conversion specification).
- 16198 If no **l** (ell) length modifier is present, characters from the input field shall be converted
16199 as if by repeated calls to the **wcrtomb()** function, with the conversion state described by
16200 an **mbstate_t** object initialized to zero before the first wide character is converted. The
16201 corresponding argument shall be a pointer to the initial element of a character array
16202 large enough to accept the sequence. No null character is added.
- 16203 If an **l** (ell) length modifier is present, the corresponding argument shall be a pointer to
16204 the initial element of an array of **wchar_t** large enough to accept the sequence. No null
16205 wide character is added.
- 16206 Otherwise, the application shall ensure that the corresponding argument is a pointer to
16207 an array of **wchar_t** large enough to accept the sequence. No null wide character is
16208 added.
- 16209 p Matches an implementation-defined set of sequences, which shall be the same as the set
16210 of sequences that is produced by the %p conversion specification of the corresponding
16211 **fwprintf()** functions. The application shall ensure that the corresponding argument is a
16212 pointer to a pointer to **void**. The interpretation of the input item is implementation-
16213 defined. If the input item is a value converted earlier during the same program
16214 execution, the pointer that results shall compare equal to that value; otherwise, the

16215		behavior of the %p conversion is undefined.
16216	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 fwscanf() 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.
16223	XSI C	Equivalent to 1c.
16224	XSI S	Equivalent to 1s.
16225	%	Matches a single '%' wide character; no conversion or assignment shall occur. The complete conversion specification shall be %%.
16227		If a conversion specification is invalid, the behavior is undefined.
16228		The conversion specifiers A, E, F, G, and X are also valid and shall be equivalent to, respectively, a, e, f, g, and x.
16230		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.
16236		Reaching the end of the string in swscanf() shall be equivalent to encountering end-of-file for fwscanf().
16238		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.
16242	CX	The fwscanf() and wscanf() 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 fgetc(), fgetwc(), fgets(), fgetws(), fread(), getc(), getwc(), getchar(), getwchar(), gets(), fscanf(), or fwscanf() using <i>stream</i> that returns data not supplied by a prior call to ungetc().
16246		RETURN VALUE
16247		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.
16252		ERRORS
16253		For the conditions under which the fwscanf() functions shall fail and may fail, refer to fgetwc().
16254		In addition, fwscanf() may fail if:
16255	XSI [EILSEQ]	Input byte sequence does not form a valid character.
16256	XSI [EINVAL]	There are insufficient arguments.

16257 EXAMPLES

16258 The call:

```
16259       int i, n; float x; char name[50];  
16260       n = wscanf(L"%d%f%s", &i, &x, name);
```

16261 with the input line:

```
16262       25 54.32E-1 Hamster
```

16263 assigns to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* contains the string "Hamster".

16265 The call:

```
16266       int i; float x; char name[50];  
16267       (void) wscanf(L"%2d%f%*d %[0123456789]", &i, &x, name);
```

16268 with input:

```
16269       56789 0123 56a72
```

16270 assigns 56 to *i*, 789.0 to *x*, skips 0123, and places the string "56\0" in *name*. The next call to *getchar()* shall return the character 'a'.

16272 APPLICATION USAGE

16273 In format strings containing the '%' form of conversion specifications, each argument in the argument list is used exactly once.

16275 RATIONALE

16276 None.

16277 FUTURE DIRECTIONS

16278 None.

16279 SEE ALSO

16280 *getwc()*, *fwprintf()*, *setlocale()*, *wcstod()*, *wcstoul()*, *wcrtomb()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, *<langinfo.h>*, *<stdio.h>*, *<wchar.h>*

16282 CHANGE HISTORY

16283 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

16285 Issue 6

16286 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

16287 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 16288 • The prototypes for *fwscanf()* and *swscanf()* are updated.
- 16289 • The DESCRIPTION is updated.
- 16290 • The hh, ll, j, t, and z length modifiers are added.
- 16291 • The a, A, and F conversion characters are added.

16292 The DESCRIPTION is updated to use the terms "conversion specifier" and "conversion specification" consistently.

16294 NAME

16295 *gai_strerror* — address and name information error description

16296 SYNOPSIS

16297 #include <netdb.h>
16298 const char *gai_strerror(int ecode);

16299 DESCRIPTION

16300 The *gai_strerror()* function shall return a text string describing an error value for the *getaddrinfo()* and *getnameinfo()* functions listed in the <netdb.h> header.

16302 When the *ecode* argument is one of the following values listed in the <netdb.h> header:

16303 [EAI_AGAIN]
16304 [EAI_BADFLAGS]
16305 [EAI_FAIL]
16306 [EAI_FAMILY]
16307 [EAI_MEMORY]
16308 [EAI_NONAME]
16309 [EAI_OVERFLOW]
16310 [EAI_SERVICE]
16311 [EAI_SOCKTYPE]
16312 [EAI_SYSTEM]

1

16313 the function return value shall point to a string describing the error. If the argument is not one
16314 of those values, the function shall return a pointer to a string whose contents indicate an
16315 unknown error.

16316 RETURN VALUE

16317 Upon successful completion, *gai_strerror()* shall return a pointer to an implementation-defined
16318 string.

16319 ERRORS

16320 No errors are defined.

16321 EXAMPLES

16322 None.

16323 APPLICATION USAGE

16324 None.

16325 RATIONALE

16326 None.

16327 FUTURE DIRECTIONS

16328 None.

16329 SEE ALSO

16330 *getaddrinfo()*, the Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>

16331 CHANGE HISTORY

16332 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

16333 The Open Group Base Resolution bwg2001-009 is applied, which changes the return type from
16334 *char ** to *const char **. This is for coordination with the IPnG Working Group.

16335 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/22 is applied, adding the 1
16336 [EAI_OVERFLOW] error code. 1

16337 NAME

16338 *gcvt* — convert a floating-point number to a string (LEGACY)

16339 SYNOPSIS

16340 XSI `#include <stdlib.h>`

16341 `char *gcvt(double value, int ndigit, char *buf);`

16342

16343 DESCRIPTION

16344 Refer to *ecvt()*.

16345 **NAME**16346 **getaddrinfo** — get address information16347 **SYNOPSIS**

```
16348     #include <sys/socket.h>
16349     #include <netdb.h>
16350     int getaddrinfo(const char *restrict nodename,
16351                     const char *restrict servname,
16352                     const struct addrinfo *restrict hints,
16353                     struct addrinfo **restrict res);
```

16354 **DESCRIPTION**16355 Refer to *freeaddrinfo()*.

16356 NAME

16357 `getc` — get a byte from a stream

16358 SYNOPSIS

16359 `#include <stdio.h>`
16360 `int getc(FILE *stream);`

16361 DESCRIPTION

16362 CX 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.

16365 The `getc()` function shall be equivalent to `fgetc()`, except that if it is implemented as a macro it
16366 may evaluate *stream* more than once, so the argument should never be an expression with side
16367 effects.

16368 RETURN VALUE

16369 Refer to `fgetc()`.

16370 ERRORS

16371 Refer to `fgetc()`.

16372 EXAMPLES

16373 None.

16374 APPLICATION USAGE

16375 If the integer value returned by `getc()` is stored into a variable of type **char** and then compared
16376 against the integer constant EOF, the comparison may never succeed, because sign-extension of
16377 a variable of type **char** on widening to integer is implementation-defined.

16378 Since it may be implemented as a macro, `getc()` may treat incorrectly a *stream* argument with
16379 side effects. In particular, `getc(*f++)` does not necessarily work as expected. Therefore, use of this
16380 function should be preceded by "#undef `getc`" in such situations; `fgetc()` could also be used.

16381 RATIONALE

16382 None.

16383 FUTURE DIRECTIONS

16384 None.

16385 SEE ALSO

16386 `fgetc()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`

16387 CHANGE HISTORY

16388 First released in Issue 1. Derived from Issue 1 of the SVID.

16389 NAME

16390 `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` — stdio with explicit client
 16391 locking

16392 SYNOPSIS

```
16393 TSF        #include <stdio.h>
16394        int getc_unlocked(FILE *stream);
16395        int getchar_unlocked(void);
16396        int putc_unlocked(int c, FILE *stream);
16397        int putchar_unlocked(int c);
16398
```

16399 DESCRIPTION

16400 Versions of the functions `getc()`, `getchar()`, `putc()`, and `putchar()` respectively named
 16401 `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, and `putchar_unlocked()` shall be provided
 16402 which are functionally equivalent to the original versions, with the exception that they are not
 16403 required to be implemented in a thread-safe manner. They may only safely be used within a
 16404 scope protected by `flockfile()` (or `ftrylockfile()`) and `funlockfile()`. These functions may safely be
 16405 used in a multi-threaded program if and only if they are called while the invoking thread owns
 16406 the `(FILE *)` object, as is the case after a successful call to the `flockfile()` or `ftrylockfile()` functions.

16407 RETURN VALUE

16408 See `getc()`, `getchar()`, `putc()`, and `putchar()`.

16409 ERRORS

16410 See `getc()`, `getchar()`, `putc()`, and `putchar()`.

16411 EXAMPLES

16412 None.

16413 APPLICATION USAGE

16414 Since they may be implemented as macros, `getc_unlocked()` and `putc_unlocked()` may treat
 16415 incorrectly a `stream` argument with side effects. In particular, `getc_unlocked(*f++)` and
 16416 `putc_unlocked(*f++)` do not necessarily work as expected. Therefore, use of these functions in
 16417 such situations should be preceded by the following statement as appropriate:

```
16418        #undef getc_unlocked
16419        #undef putc_unlocked
```

16420 RATIONALE

16421 Some I/O functions are typically implemented as macros for performance reasons (for example,
 16422 `putc()` and `getc()`). For safety, they need to be synchronized, but it is often too expensive to
 16423 synchronize on every character. Nevertheless, it was felt that the safety concerns were more
 16424 important; consequently, the `getc()`, `getchar()`, `putc()`, and `putchar()` functions are required to be
 16425 thread-safe. However, unlocked versions are also provided with names that clearly indicate the
 16426 unsafe nature of their operation but can be used to exploit their higher performance. These
 16427 unlocked versions can be safely used only within explicitly locked program regions, using
 16428 exported locking primitives. In particular, a sequence such as:

```
16429        flockfile(fileptr);
16430        putc_unlocked('1', fileptr);
16431        putc_unlocked('\n', fileptr);
16432        fprintf(fileptr, "Line 2\n");
16433        funlockfile(fileptr);
```

16434 is permissible, and results in the text sequence:

16435 1
16436 Line 2
16437 being printed without being interspersed with output from other threads.
16438 It would be wrong to have the standard names such as *getc()*, *putc()*, and so on, map to the
16439 “faster, but unsafe” rather than the “slower, but safe” versions. In either case, you would still
16440 want to inspect all uses of *getc()*, *putc()*, and so on, by hand when converting existing code.
16441 Choosing the safe bindings as the default, at least, results in correct code and maintains the
16442 “atomicity at the function” invariant. To do otherwise would introduce gratuitous
16443 synchronization errors into converted code. Other routines that modify the *stdio* (*FILE **)
16444 structures or buffers are also safely synchronized.
16445 Note that there is no need for functions of the form *getc_locked()*, *putc_locked()*, and so on, since
16446 this is the functionality of *getc()*, *putc()*, *et al.* It would be inappropriate to use a feature test
16447 macro to switch a macro definition of *getc()* between *getc_locked()* and *getc_unlocked()*, since the
16448 ISO C standard requires an actual function to exist, a function whose behavior could not be
16449 changed by the feature test macro. Also, providing both the *xxx_locked()* and *xxx_unlocked()*
16450 forms leads to the confusion of whether the suffix describes the behavior of the function or the
16451 circumstances under which it should be used.
16452 Three additional routines, *flockfile()*, *ftrylockfile()*, and *funlockfile()* (which may be macros), are
16453 provided to allow the user to delineate a sequence of I/O statements that are executed
16454 synchronously.
16455 The *ungetc()* function is infrequently called relative to the other functions/macros so no
16456 unlocked variation is needed.

16457 **FUTURE DIRECTIONS**
16458 None.

16459 **SEE ALSO**
16460 *getc()*, *getchar()*, *putc()*, *putchar()*, the Base Definitions volume of IEEE Std 1003.1-2001,
16461 *<stdio.h>*

16462 **CHANGE HISTORY**
16463 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

16464 **Issue 6**
16465 These functions are marked as part of the Thread-Safe Functions option.
16466 The Open Group Corrigendum U030/2 is applied, adding APPLICATION USAGE describing
16467 how applications should be written to avoid the case when the functions are implemented as
16468 macros.

16469 NAME

16470 *getchar* — get a byte from a stdin stream

16471 SYNOPSIS

```
16472        #include <stdio.h>
16473        int getchar(void);
```

16474 DESCRIPTION

16475 CX The functionality described on this reference page is aligned with the ISO C standard. Any
16476 conflict between the requirements described here and the ISO C standard is unintentional. This
16477 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

16478 The *getchar*() function shall be equivalent to *getc(stdin)*.

16479 RETURN VALUE

16480 Refer to *fgetc*().

16481 ERRORS

16482 Refer to *fgetc*().

16483 EXAMPLES

16484 None.

16485 APPLICATION USAGE

16486 If the integer value returned by *getchar*() is stored into a variable of type **char** and then
16487 compared against the integer constant EOF, the comparison may never succeed, because sign-
16488 extension of a variable of type **char** on widening to integer is implementation-defined.

16489 RATIONALE

16490 None.

16491 FUTURE DIRECTIONS

16492 None.

16493 SEE ALSO

16494 *getc*(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

16495 CHANGE HISTORY

16496 First released in Issue 1. Derived from Issue 1 of the SVID.

16497 NAME

16498 `getchar_unlocked` — stdio with explicit client locking

16499 SYNOPSIS

16500 TSF `#include <stdio.h>`

16501 `int getchar_unlocked(void);`

16502

16503 DESCRIPTION

16504 Refer to *getc_unlocked()*.

16505 NAME

16506 getcontext, setcontext — get and set current user context

16507 SYNOPSIS

16508 OB XSI #include <ucontext.h>

2

16509 int getcontext(ucontext_t *ucp);
16510 int setcontext(const ucontext_t *ucp);
16511

16512 DESCRIPTION

16513 The *getcontext()* function shall initialize the structure pointed to by *ucp* to the current user
16514 context of the calling thread. The **ucontext_t** type that *ucp* points to defines the user context and
16515 includes the contents of the calling thread's machine registers, the signal mask, and the current
16516 execution stack.

16517 The *setcontext()* function shall restore the user context pointed to by *ucp*. A successful call to
16518 *setcontext()* shall not return; program execution resumes at the point specified by the *ucp*
16519 argument passed to *setcontext()*. The *ucp* argument should be created either by a prior call to
16520 *getcontext()* or *makecontext()*, or by being passed as an argument to a signal handler. If the *ucp*
16521 argument was created with *getcontext()*, program execution continues as if the corresponding
16522 call of *getcontext()* had just returned. If the *ucp* argument was created with *makecontext()*,
16523 program execution continues with the function passed to *makecontext()*. When that function
16524 returns, the thread shall continue as if after a call to *setcontext()* with the *ucp* argument that was
16525 input to *makecontext()*. If the *uc_link* member of the **ucontext_t** structure pointed to by the *ucp*
16526 argument is equal to 0, then this context is the main context, and the thread shall exit when this
16527 context returns. The effects of passing a *ucp* argument obtained from any other source are
16528 unspecified.

16529 RETURN VALUE

16530 Upon successful completion, *setcontext()* shall not return and *getcontext()* shall return 0;
16531 otherwise, a value of -1 shall be returned.

16532 ERRORS

16533 No errors are defined.

16534 EXAMPLES

16535 Refer to *makecontext()*.

16536 APPLICATION USAGE

16537 When a signal handler is executed, the current user context is saved and a new context is
16538 created. If the thread leaves the signal handler via *longjmp()*, then it is unspecified whether the
16539 context at the time of the corresponding *setjmp()* call is restored and thus whether future calls to
16540 *getcontext()* provide an accurate representation of the current context, since the context restored
16541 by *longjmp()* does not necessarily contain all the information that *setcontext()* requires. Signal
16542 handlers should use *siglongjmp()* or *setcontext()* instead.

16543 Conforming applications should not modify or access the *uc_mcontext* member of **ucontext_t**. A
16544 conforming application cannot assume that context includes any process-wide static data,
16545 possibly including *errno*. Users manipulating contexts should take care to handle these
16546 explicitly when required.

16547 Use of contexts to create alternate stacks is not defined by this volume of IEEE Std 1003.1-2001.

16548 The obsolescent functions *getcontext()*, *makecontext()*, and *swapcontext()* can be replaced using
16549 POSIX threads functions.

2
2

16550 RATIONALE

16551 None.

16552 FUTURE DIRECTIONS

16553 None.

16554 SEE ALSO

16555 *bsd_signal()*, *makecontext()*, *setcontext()*, *setjmp()*, *sigaction()*, *sigaltstack()*, *siglongjmp()*,
16556 *sigprocmask()*, *sigsetjmp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <ucontext.h>

16557 CHANGE HISTORY

16558 First released in Issue 4, Version 2.

16559 Issue 5

16560 Moved from X/OPEN UNIX extension to BASE.

16561 The following sentence was removed from the DESCRIPTION: ‘If the *ucp* argument was passed
16562 to a signal handler, program execution continues with the program instruction following the
16563 instruction interrupted by the signal.’

16564 Issue 6

16565 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/45 is applied, updating the SYNOPSIS 2
16566 and APPLICATION USAGE sections to note that the *getcontext()* and *setcontext()* functions are 2
16567 obsolescent. 2

16568 **NAME**

16569 `getcwd` — get the pathname of the current working directory

16570 **SYNOPSIS**

```
16571        #include <unistd.h>
16572        char *getcwd(char *buf, size_t size);
```

16573 **DESCRIPTION**

16574 The `getcwd()` function shall place an absolute pathname of the current working directory in the
16575 array pointed to by `buf`, and return `buf`. The pathname copied to the array shall contain no
16576 components that are symbolic links. The `size` argument is the size in bytes of the character array
16577 pointed to by the `buf` argument. If `buf` is a null pointer, the behavior of `getcwd()` is unspecified.

16578 **RETURN VALUE**

16579 Upon successful completion, `getcwd()` shall return the `buf` argument. Otherwise, `getcwd()` shall
16580 return a null pointer and set `errno` to indicate the error. The contents of the array pointed to by
16581 `buf` are then undefined.

16582 **ERRORS**

16583 The `getcwd()` function shall fail if:

16584 [EINVAL] The `size` argument is 0.

16585 [ERANGE] The `size` argument is greater than 0, but is smaller than the length of the
16586 pathname +1.

16587 The `getcwd()` function may fail if:

16588 [EACCES] Read or search permission was denied for a component of the pathname.

16589 [ENOMEM] Insufficient storage space is available.

16590 **EXAMPLES**16591 **Determining the Absolute Pathname of the Current Working Directory**

16592 The following example returns a pointer to an array that holds the absolute pathname of the
16593 current working directory. The pointer is returned in the `ptr` variable, which points to the `buf`
16594 array where the pathname is stored.

```
16595        #include <stdlib.h>
16596        #include <unistd.h>
16597        ...
16598        long size;
16599        char *buf;
16600        char *ptr;
16601        size = pathconf(".", _PC_PATH_MAX);
16602        if ((buf = (char *)malloc((size_t)size)) != NULL)
16603            ptr = getcwd(buf, (size_t)size);
16604            ...
```

16605 **APPLICATION USAGE**

16606 None.

16607 RATIONALE

16608 Since the maximum pathname length is arbitrary unless {PATH_MAX} is defined, an application
16609 generally cannot supply a *buf* with size {PATH_MAX}+1.

16610 Having *getcwd()* take no arguments and instead use the *malloc()* function to produce space for
16611 the returned argument was considered. The advantage is that *getcwd()* knows how big the
16612 working directory pathname is and can allocate an appropriate amount of space. But the
16613 programmer would have to use the *free()* function to free the resulting object, or each use of
16614 *getcwd()* would further reduce the available memory. Also, *malloc()* and *free()* are used nowhere
16615 else in this volume of IEEE Std 1003.1-2001. Finally, *getcwd()* is taken from the SVID where it has
16616 the two arguments used in this volume of IEEE Std 1003.1-2001.

16617 The older function *getwd()* was rejected for use in this context because it had only a buffer
16618 argument and no *size* argument, and thus had no way to prevent overwriting the buffer, except
16619 to depend on the programmer to provide a large enough buffer.

16620 On some implementations, if *buf* is a null pointer, *getcwd()* may obtain *size* bytes of memory
16621 using *malloc()*. In this case, the pointer returned by *getcwd()* may be used as the argument in a
16622 subsequent call to *free()*. Invoking *getcwd()* with *buf* as a null pointer is not recommended in
16623 conforming applications.

16624 If a program is operating in a directory where some (grand)parent directory does not permit
16625 reading, *getcwd()* may fail, as in most implementations it must read the directory to determine
16626 the name of the file. This can occur if search, but not read, permission is granted in an
16627 intermediate directory, or if the program is placed in that directory by some more privileged
16628 process (for example, login). Including the [EACCES] error condition makes the reporting of the
16629 error consistent and warns the application writer that *getcwd()* can fail for reasons beyond the
16630 control of the application writer or user. Some implementations can avoid this occurrence (for
16631 example, by implementing *getcwd()* using *pwd*, where *pwd* is a set-user-root process), thus the
16632 error was made optional. Since this volume of IEEE Std 1003.1-2001 permits the addition of other
16633 errors, this would be a common addition and yet one that applications could not be expected to
16634 deal with without this addition.

16635 FUTURE DIRECTIONS

16636 None.

16637 SEE ALSO

16638 *malloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

16639 CHANGE HISTORY

16640 First released in Issue 1. Derived from Issue 1 of the SVID.

16641 Issue 6

16642 The following new requirements on POSIX implementations derive from alignment with the
16643 Single UNIX Specification:

- 16644 • The [ENOMEM] optional error condition is added.

16645 NAME

16646 getdate — convert user format date and time

16647 SYNOPSIS

16648 XSI #include <time.h>

16649 struct tm *getdate(const char *string);

16650

16651 DESCRIPTION

16652 The *getdate()* function shall convert a string representation of a date or time into a broken-down
16653 time.

16654 The external variable or macro *getdate_err* is used by *getdate()* to return error values.

16655 Templates are used to parse and interpret the input string. The templates are contained in a text
16656 file identified by the environment variable *DATEMSK*. The *DATEMSK* variable should be set to
16657 indicate the full pathname of the file that contains the templates. The first line in the template
16658 that matches the input specification is used for interpretation and conversion into the internal
16659 time format.

16660 The following conversion specifications shall be supported:

16661 %% Equivalent to %.

16662 %a Abbreviated weekday name.

16663 %A Full weekday name.

16664 %b Abbreviated month name.

16665 %B Full month name.

16666 %c Locale's appropriate date and time representation.

16667 %C Century number [00,99]; leading zeros are permitted but not required.

16668 %d Day of month [01,31]; the leading 0 is optional.

16669 %D Date as %m/%d/%y.

16670 %e Equivalent to %d.

16671 %h Abbreviated month name.

16672 %H Hour [00,23].

16673 %I Hour [01,12].

16674 %m Month number [01,12].

16675 %M Minute [00,59].

16676 %n Equivalent to <newline>.

16677 %p Locale's equivalent of either AM or PM.

16678 %r The locale's appropriate representation of time in AM and PM notation. In the POSIX
16679 locale, this shall be equivalent to %I:%M:%S %p.

16680 %R Time as %H:%M.

16681 %S Seconds [00,60]. The range goes to 60 (rather than stopping at 59) to allow positive leap
16682 seconds to be expressed. Since leap seconds cannot be predicted by any algorithm, leap
16683 second data must come from some external source.

16684	%t	Equivalent to <tab>.
16685	%T	Time as %H:%M:%S.
16686	%w	Weekday number (Sunday = [0,6]).
16687	%x	Locale's appropriate date representation.
16688	%X	Locale's appropriate time representation.
16689	%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.
16692	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.)
16695	%Y	Year as "ccyy" (for example, 2001).
16696	%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).
16700		The match between the template and input specification performed by <i>getdate()</i> shall be case-insensitive.
16702		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>).
16705		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.
16708		The results are undefined if the conversion specifications %c, %x, and %X include unsupported conversion specifications.
16710		The following rules apply for converting the input specification into the internal format:
16711		• 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.
16714		• 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.
16716		• 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.
16719		• If no hour, minute, and second are given, the current hour, minute, and second shall be assumed.
16721		• 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.
16723		If a conversion specification in the DATEMSK file does not correspond to one of the conversion specifications above, the behavior is unspecified.
16725		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.

16727 RETURN VALUE

16728 Upon successful completion, *getdate()* shall return a pointer to a **struct tm**. Otherwise, it shall
16729 return a null pointer and set *getdate_err* to indicate the error.

16730 ERRORS

16731 The *getdate()* function shall fail in the following cases, setting *getdate_err* to the value shown in
16732 the list below. Any changes to *errno* are unspecified.

- 16733 1. The *DATEMSK* environment variable is null or undefined.
- 16734 2. The template file cannot be opened for reading.
- 16735 3. Failed to get file status information.
- 16736 4. The template file is not a regular file.
- 16737 5. An I/O error is encountered while reading the template file.
- 16738 6. Memory allocation failed (not enough memory available).
- 16739 7. There is no line in the template that matches the input.
- 16740 8. Invalid input specification. For example, February 31; or a time is specified that cannot be
16741 represented in a **time_t** (representing the time in seconds since the Epoch).

16742 EXAMPLES

- 16743 1. The following example shows the possible contents of a template:

```
16744       %m
16745       %A %B %d, %Y, %H:%M:%S
16746       %A
16747       %B
16748       %m/%d/%Y %I %p
16749       %d,%m,%Y %H:%M
16750       at %A the %dst of %B in %Y
16751       run job at %I %p,%B %dnd
16752       %A den %d. %B %Y %H.%M Uhr
```

- 16753 2. The following are examples of valid input specifications for the template in Example 1:

```
16754       getdate( "10/1/87 4 PM" );
16755       getdate( "Friday" );
16756       getdate( "Friday September 18, 1987, 10:30:30" );
16757       getdate( "24,9,1986 10:30" );
16758       getdate( "at monday the 1st of december in 1986" );
16759       getdate( "run job at 3 PM, december 2nd" );
```

16760 If the *LC_TIME* category is set to a German locale that includes *freitag* as a weekday name
16761 and *oktober* as a month name, the following would be valid:

```
16762       getdate( "freitag den 10. oktober 1986 10.30 Uhr" );
```

- 16763 3. The following example shows how local date and time specification can be defined in the
16764 template:

16765
16766
16767
16768
16769
16770

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

- 16771 4. The following examples help to illustrate the above rules assuming that the current date is
 16772 Mon Sep 22 12:19:47 EDT 1986 and the *LC_TIME* category is set to the default C locale:

16773
16774
16775
16776
16777
16778
16779
16780
16781
16782
16783
16784
16785
16786
16787
16788

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

16789 APPLICATION USAGE

16790 Although historical versions of *getdate()* did not require that <time.h> declare the external
 16791 variable *getdate_err*, this volume of IEEE Std 1003.1-2001 does require it. The standard
 16792 developers encourage applications to remove declarations of *getdate_err* and instead incorporate
 16793 the declaration by including <time.h>.

16794 Applications should use %Y (4-digit years) in preference to %y (2-digit years).

16795 RATIONALE

16796 In standard locales, the conversion specifications %C, %x, and %X do not include unsupported
 16797 conversion specifiers and so the text regarding results being undefined is not a problem in that
 16798 case.

16799 FUTURE DIRECTIONS

16800 None.

16801 SEE ALSO

16802 *ctime()*, *localtime()*, *setlocale()*, *strftime()*, *times()*, the Base Definitions volume of
 16803 IEEE Std 1003.1-2001, <time.h>

16804 CHANGE HISTORY

16805 First released in Issue 4, Version 2.

16806 Issue 5

16807 Moved from X/OPEN UNIX extension to BASE.

16808 The last paragraph of the DESCRIPTION is added.

16809 The %C conversion specification is added, and the exact meaning of the %y conversion
 16810 specification is clarified in the DESCRIPTION.

- 16811 A note indicating that this function need not be reentrant is added to the DESCRIPTION.
- 16812 The %R conversion specification is changed to follow historical practice.
- 16813 **Issue 6**
- 16814 The DESCRIPTION is updated to refer to “seconds since the Epoch” rather than “seconds since 00:00:00 UTC (Coordinated Universal Time), January 1 1970” for consistency with other *time* functions.
- 16815
- 16816
- 16817 The description of %S is updated so that the valid range is [00,60] rather than [00,61].
- 16818 The DESCRIPTION is updated to refer to conversion specifications instead of field descriptors for consistency with other functions.
- 16819

16820 NAME

16821 *getegid* — get the effective group ID

16822 SYNOPSIS

```
16823        #include <unistd.h>
16824        gid_t getegid(void);
```

16825 DESCRIPTION

16826 The *getegid*() function shall return the effective group ID of the calling process.

16827 RETURN VALUE

16828 The *getegid*() function shall always be successful and no return value is reserved to indicate an error.

16830 ERRORS

16831 No errors are defined.

16832 EXAMPLES

16833 None.

16834 APPLICATION USAGE

16835 None.

16836 RATIONALE

16837 None.

16838 FUTURE DIRECTIONS

16839 None.

16840 SEE ALSO

16841 *geteuid*(), *getgid*(), *getuid*(), *setegid*(), *seteuid*(), *setgid*(), *setregid*(), *setreuid*(), *setuid*(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

16843 CHANGE HISTORY

16844 First released in Issue 1. Derived from Issue 1 of the SVID.

16845 Issue 6

16846 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

16847 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- 16849 • 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.
- 16850
- 16851

16852 NAME

16853 getenv — get value of an environment variable

16854 SYNOPSIS

```
16855 #include <stdlib.h>
16856 char *getenv(const char *name);
```

16857 DESCRIPTION

16858 CX The functionality described on this reference page is aligned with the ISO C standard. Any
16859 conflict between the requirements described here and the ISO C standard is unintentional. This
16860 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

16861 The *getenv()* function shall search the environment of the calling process (see the Base
16862 Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables) for the
16863 environment variable *name* if it exists and return a pointer to the value of the environment
16864 variable. If the specified environment variable cannot be found, a null pointer shall be returned.
16865 The application shall ensure that it does not modify the string pointed to by the *getenv()*
16866 function.

16867 CX The string pointed to may be overwritten by a subsequent call to *getenv()*, *setenv()*, or *unsetenv()*,
16868 but shall not be overwritten by a call to any other function in this volume of
16869 IEEE Std 1003.1-2001.

16870 CX If the application modifies *environ* or the pointers to which it points, the behavior of *getenv()* is
16871 undefined.

16872 The *getenv()* function need not be reentrant. A function that is not required to be reentrant is not
16873 required to be thread-safe.

16874 RETURN VALUE

16875 Upon successful completion, *getenv()* shall return a pointer to a string containing the *value* for
16876 the specified *name*. If the specified *name* cannot be found in the environment of the calling
16877 process, a null pointer shall be returned.

16878 The return value from *getenv()* may point to static data which may be overwritten by
16879 CX subsequent calls to *getenv()*, *setenv()*, or *unsetenv()*.

16880 XSI On XSI-conformant systems, the return value from *getenv()* may point to static data which may
16881 also be overwritten by subsequent calls to *putenv()*.

16882 ERRORS

16883 No errors are defined.

16884 EXAMPLES

16885 Getting the Value of an Environment Variable

16886 The following example gets the value of the *HOME* environment variable.

```
16887 #include <stdlib.h>
16888 ...
16889 const char *name = "HOME";
16890 char *value;
16891 value = getenv(name);
```

16892 APPLICATION USAGE

16893 None.

16894 RATIONALE

16895 The *clearenv()* function was considered but rejected. The *putenv()* function has now been
16896 included for alignment with the Single UNIX Specification.16897 The *getenv()* function is inherently not reentrant because it returns a value pointing to static
16898 data.16899 Conforming applications are required not to modify *environ* directly, but to use only the
16900 functions described here to manipulate the process environment as an abstract object. Thus, the
16901 implementation of the environment access functions has complete control over the data
16902 structure used to represent the environment (subject to the requirement that *environ* be
16903 maintained as a list of strings with embedded equal signs for applications that wish to scan the
16904 environment). This constraint allows the implementation to properly manage the memory it
16905 allocates, either by using allocated storage for all variables (copying them on the first invocation
16906 of *setenv()* or *unsetenv()*), or keeping track of which strings are currently in allocated space and
16907 which are not, via a separate table or some other means. This enables the implementation to free
16908 any allocated space used by strings (and perhaps the pointers to them) stored in *environ* when
16909 *unsetenv()* is called. A C runtime start-up procedure (that which invokes *main()* and perhaps
16910 initializes *environ*) can also initialize a flag indicating that none of the environment has yet been
16911 copied to allocated storage, or that the separate table has not yet been initialized.16912 In fact, for higher performance of *getenv()*, the implementation could also maintain a separate
16913 copy of the environment in a data structure that could be searched much more quickly (such as
16914 an indexed hash table, or a binary tree), and update both it and the linear list at *environ* when
16915 *setenv()* or *unsetenv()* is invoked.16916 Performance of *getenv()* can be important for applications which have large numbers of
16917 environment variables. Typically, applications like this use the environment as a resource
16918 database of user-configurable parameters. The fact that these variables are in the user's shell
16919 environment usually means that any other program that uses environment variables (such as *ls*,
16920 which attempts to use *COLUMNS*, or really almost any utility (*LANG*, *LC_ALL*, and so on) is
16921 similarly slowed down by the linear search through the variables.16922 An implementation that maintains separate data structures, or even one that manages the
16923 memory it consumes, is not currently required as it was thought it would reduce consensus
16924 among implementors who do not want to change their historical implementations.16925 The POSIX Threads Extension states that multi-threaded applications must not modify *environ*
16926 directly, and that IEEE Std 1003.1-2001 is providing functions which such applications can use in
16927 the future to manipulate the environment in a thread-safe manner. Thus, moving away from
16928 application use of *environ* is desirable from that standpoint as well.

16929 FUTURE DIRECTIONS

16930 None.

16931 SEE ALSO

16932 *exec*, *putenv()*, *setenv()*, *unsetenv()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter
16933 8, Environment Variables, <stdlib.h>

16934 CHANGE HISTORY

16935 First released in Issue 1. Derived from Issue 1 of the SVID.

16936 Issue 5

16937 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
16938 VALUE section.

16939 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

16940 Issue 6

16941 The following changes were made to align with the IEEE P1003.1a draft standard:

- References added to the new *setenv()* and *unsetenv()* functions.

16943 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

16944 NAME

16945 *geteuid* — get the effective user ID

16946 SYNOPSIS

```
16947        #include <unistd.h>
16948        uid_t geteuid(void);
```

16949 DESCRIPTION

16950 The *geteuid()* function shall return the effective user ID of the calling process.

16951 RETURN VALUE

16952 The *geteuid()* function shall always be successful and no return value is reserved to indicate an error.

16954 ERRORS

16955 No errors are defined.

16956 EXAMPLES

16957 None.

16958 APPLICATION USAGE

16959 None.

16960 RATIONALE

16961 None.

16962 FUTURE DIRECTIONS

16963 None.

16964 SEE ALSO

16965 *getegid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**sys/types.h**>, <**unistd.h**>

16967 CHANGE HISTORY

16968 First released in Issue 1. Derived from Issue 1 of the SVID.

16969 Issue 6

16970 In the SYNOPSIS, the optional include of the <**sys/types.h**> header is removed.

16971 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- 16973 • 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.
- 16974
- 16975

16976 NAME

16977 *getgid* — get the real group ID

16978 SYNOPSIS

```
16979        #include <unistd.h>
16980        gid_t getgid(void);
```

16981 DESCRIPTION

16982 The *getgid()* function shall return the real group ID of the calling process.

16983 RETURN VALUE

16984 The *getgid()* function shall always be successful and no return value is reserved to indicate an error.

16986 ERRORS

16987 No errors are defined.

16988 EXAMPLES

16989 None.

16990 APPLICATION USAGE

16991 None.

16992 RATIONALE

16993 None.

16994 FUTURE DIRECTIONS

16995 None.

16996 SEE ALSO

16997 *getegid()*, *geteuid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

16999 CHANGE HISTORY

17000 First released in Issue 1. Derived from Issue 1 of the SVID.

17001 Issue 6

17002 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

17003 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- 17005 • 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.
- 17006
- 17007

17008 **NAME**

17009 getgrent — get the group database entry

17010 **SYNOPSIS**

17011 XSI #include <grp.h>

17012 struct group *getgrent(void);

17013

17014 **DESCRIPTION**

17015 Refer to *endgrent()*.

17016 NAME

17017 `getgrgid, getgrgid_r — get group database entry for a group ID`

17018 SYNOPSIS

```
17019        #include <grp.h>
17020
17021        struct group *getgrgid(gid_t gid);
17022        TSF      int getgrgid_r(gid_t gid, struct group *grp, char *buffer,
17023                          size_t bufsize, struct group **result);
17024
```

17024 DESCRIPTION

17025 The `getgrgid()` function shall search the group database for an entry with a matching `gid`.

17026 The `getgrgid()` function need not be reentrant. A function that is not required to be reentrant is
17027 not required to be thread-safe.

17028 TSF The `getgrgid_r()` function shall update the **group** structure pointed to by `grp` and store a pointer
17029 to that structure at the location pointed to by `result`. The structure shall contain an entry from
17030 the group database with a matching `gid`. Storage referenced by the group structure is allocated
17031 from the memory provided with the `buffer` parameter, which is `bufsize` bytes in size. The
17032 maximum size needed for this buffer can be determined with the `{_SC_GETGR_R_SIZE_MAX}`
17033 `sysconf()` parameter. A NULL pointer shall be returned at the location pointed to by `result` on
17034 error or if the requested entry is not found.

17035 RETURN VALUE

17036 Upon successful completion, `getgrgid()` shall return a pointer to a **struct group** with the structure
17037 defined in `<grp.h>` with a matching entry if one is found. The `getgrgid()` function shall return a
17038 null pointer if either the requested entry was not found, or an error occurred. On error, `errno`
17039 shall be set to indicate the error.

17040 The return value may point to a static area which is overwritten by a subsequent call to
17041 `getrent()`, `getgrgid()`, or `getgrnam()`.

17042 TSF If successful, the `getgrgid_r()` function shall return zero; otherwise, an error number shall be
17043 returned to indicate the error.

17044 ERRORS

17045 The `getgrgid()` and `getgrgid_r()` functions may fail if:

17046	[EIO]	An I/O error has occurred.
17047	[EINTR]	A signal was caught during <code>getgrgid()</code> .
17048	[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
17049	[ENFILE]	The maximum allowable number of files is currently open in the system.
17050 TSF	The <code>getgrgid_r()</code> function may fail if:	
17051 TSF	[ERANGE]	Insufficient storage was supplied via <code>buffer</code> and <code>bufsize</code> to contain the data to 17052 be referenced by the resulting group structure.

17053 EXAMPLES

17054 Finding an Entry in the Group Database

17055 The following example uses *getgrgid()* to search the group database for a group ID that was
17056 previously stored in a **stat** structure, then prints out the group name if it is found. If the group is
17057 not found, the program prints the numeric value of the group for the entry.

```
17058 #include <sys/types.h>
17059 #include <grp.h>
17060 #include <stdio.h>
17061 ...
17062 struct stat statbuf;
17063 struct group *grp;
17064 ...
17065 if ((grp = getgrgid(statbuf.st_gid)) != NULL)
17066     printf(" %-8.8s", grp->gr_name);
17067 else
17068     printf(" %-8d", statbuf.st_gid);
17069 ...
```

17070 APPLICATION USAGE

17071 Applications wishing to check for error situations should set *errno* to 0 before calling *getgrgid()*.
17072 If *errno* is set on return, an error occurred.

17073 The *getgrgid_r()* function is thread-safe and shall return values in a user-supplied buffer instead
17074 of possibly using a static data area that may be overwritten by each call.

17075 RATIONALE

17076 None.

17077 FUTURE DIRECTIONS

17078 None.

17079 SEE ALSO

17080 *endgrent()*, *getgrnam()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<grp.h>**,
17081 **<limits.h>**, **<sys/types.h>**

17082 CHANGE HISTORY

17083 First released in Issue 1. Derived from System V Release 2.0.

17084 Issue 5

17085 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
17086 VALUE section.

17087 The *getgrgid_r()* function is included for alignment with the POSIX Threads Extension.

17088 A note indicating that the *getgrgid()* function need not be reentrant is added to the
17089 DESCRIPTION.

17090 Issue 6

17091 The *getgrgid_r()* function is marked as part of the Thread-Safe Functions option.

17092 The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION
17093 describing matching the *gid*.

17094 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

17095 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

17096 The following new requirements on POSIX implementations derive from alignment with the
17097 Single UNIX Specification:

- 17098 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
17099 required for conforming implementations of previous POSIX specifications, it was not
17100 required for UNIX applications.
- 17101 • In the RETURN VALUE section, the requirement to set *errno* on error is added.
- 17102 • The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

17103 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
17104 its avoidance of possibly using a static data area.

17105 IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the
17106 buffer from *bufsize* characters to bytes.

17107 NAME

17108 `getgrnam, getgrnam_r — search group database for a name`

17109 SYNOPSIS

17110 `#include <grp.h>`

17111 `struct group *getgrnam(const char *name);`
17112 TSF `int getgrnam_r(const char *name, struct group *grp, char *buffer,`
17113 `size_t bufsize, struct group **result);`

17115 DESCRIPTION

17116 The `getgrnam()` function shall search the group database for an entry with a matching *name*.

17117 The `getgrnam()` function need not be reentrant. A function that is not required to be reentrant is
17118 not required to be thread-safe.

17119 TSF The `getgrnam_r()` function shall update the **group** structure pointed to by *grp* and store a pointer
17120 to that structure at the location pointed to by *result*. The structure shall contain an entry from
17121 the group database with a matching *gid* or *name*. Storage referenced by the **group** structure is
17122 allocated from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
17123 maximum size needed for this buffer can be determined with the `{_SC_GETGR_R_SIZE_MAX}`
17124 `sysconf()` parameter. A NULL pointer is returned at the location pointed to by *result* on error or if
17125 the requested entry is not found.

17126 RETURN VALUE

17127 The `getgrnam()` function shall return a pointer to a **struct group** with the structure defined in
17128 `<grp.h>` with a matching entry if one is found. The `getgrnam()` function shall return a null
17129 pointer if either the requested entry was not found, or an error occurred. On error, *errno* shall be
17130 set to indicate the error.

17131 The return value may point to a static area which is overwritten by a subsequent call to
17132 `getrent()`, `getgrgid()`, or `getgrnam()`.

17133 TSF If successful, the `getgrnam_r()` function shall return zero; otherwise, an error number shall be
17134 returned to indicate the error.

17135 ERRORS

17136 The `getgrnam()` and `getgrnam_r()` functions may fail if:

17137 [EIO] An I/O error has occurred.

17138 [EINTR] A signal was caught during `getgrnam()`.

17139 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

17140 [ENFILE] The maximum allowable number of files is currently open in the system.

17141 The `getgrnam_r()` function may fail if:

17142 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
17143 be referenced by the resulting **group** structure.

17144 EXAMPLES

17145 None.

17146 APPLICATION USAGE

17147 Applications wishing to check for error situations should set *errno* to 0 before calling *getgrnam()*.
17148 If *errno* is set on return, an error occurred.

17149 The *getgrnam_r()* function is thread-safe and shall return values in a user-supplied buffer instead
17150 of possibly using a static data area that may be overwritten by each call.

17151 RATIONALE

17152 None.

17153 FUTURE DIRECTIONS

17154 None.

17155 SEE ALSO

17156 *endgrent()*, *getgrgid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <grp.h>, <limits.h>,
17157 <sys/types.h>

17158 CHANGE HISTORY

17159 First released in Issue 1. Derived from System V Release 2.0.

17160 Issue 5

17161 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
17162 VALUE section.

17163 The *getgrnam_r()* function is included for alignment with the POSIX Threads Extension.

17164 A note indicating that the *getgrnam()* function need not be reentrant is added to the
17165 DESCRIPTION.

17166 Issue 6

17167 The *getgrnam_r()* function is marked as part of the Thread-Safe Functions option.

17168 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

17169 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

17170 The following new requirements on POSIX implementations derive from alignment with the
17171 Single UNIX Specification:

- 17172 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
17173 required for conforming implementations of previous POSIX specifications, it was not
17174 required for UNIX applications.

- 17175 • In the RETURN VALUE section, the requirement to set *errno* on error is added.

- 17176 • The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

17177 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
17178 its avoidance of possibly using a static data area.

17179 IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the
17180 buffer from *bufsize* characters to bytes.

17181 NAME

17182 getgroups — get supplementary group IDs

17183 SYNOPSIS

```
17184     #include <unistd.h>
17185     int getgroups(int gidsetsize, gid_t grouplist[]);
```

17186 DESCRIPTION

17187 The *getgroups()* function shall fill in the array *grouplist* with the current supplementary group
17188 IDs of the calling process. It is implementation-defined whether *getgroups()* also returns the
17189 effective group ID in the *grouplist* array.

17190 The *gidsetsize* argument specifies the number of elements in the array *grouplist*. The actual
17191 number of group IDs stored in the array shall be returned. The values of array entries with
17192 indices greater than or equal to the value returned are undefined.

17193 If *gidsetsize* is 0, *getgroups()* shall return the number of group IDs that it would otherwise return
17194 without modifying the array pointed to by *grouplist*.

17195 If the effective group ID of the process is returned with the supplementary group IDs, the value
17196 returned shall always be greater than or equal to one and less than or equal to the value of
17197 {NGROUPS_MAX}+1.

17198 RETURN VALUE

17199 Upon successful completion, the number of supplementary group IDs shall be returned. A
17200 return value of -1 indicates failure and *errno* shall be set to indicate the error.

17201 ERRORS

17202 The *getgroups()* function shall fail if:

17203 [EINVAL] The *gidsetsize* argument is non-zero and less than the number of group IDs
17204 that would have been returned.

17205 EXAMPLES**17206 Getting the Supplementary Group IDs of the Calling Process**

17207 The following example places the current supplementary group IDs of the calling process into
17208 the *group* array.

```
17209     #include <sys/types.h>
17210     #include <unistd.h>
17211     ...
17212     gid_t *group;
17213     int nogroups;
17214     long ngroups_max;
17215
17216     ngroups_max = sysconf(_SC_NGROUPS_MAX) + 1;
17217     group = (gid_t *)malloc(ngroups_max * sizeof(gid_t));
17218
17219     nogroups = getgroups(ngroups_max, group);
```

17218 APPLICATION USAGE

17219 None.

17220 RATIONALE

17221 The related function *setgroups()* is a privileged operation and therefore is not covered by this
17222 volume of IEEE Std 1003.1-2001.

As implied by the definition of supplementary groups, the effective group ID may appear in the array returned by *getgroups()* or it may be returned only by *getegid()*. Duplication may exist, but the application needs to call *getegid()* to be sure of getting all of the information. Various implementation variations and administrative sequences cause the set of groups appearing in the result of *getgroups()* to vary in order and as to whether the effective group ID is included, even when the set of groups is the same (in the mathematical sense of “set”). (The history of a process and its parents could affect the details of the result.)

Application writers should note that {NGROUPS_MAX} is not necessarily a constant on all implementations.

17232 FUTURE DIRECTIONS

17233 None.

17234 SEE ALSO

17235 *getegid()*, *setgid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>,
17236 <unistd.h>

17237 CHANGE HISTORY

17238 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

17239 Issue 5

17240 Normative text previously in the APPLICATION USAGE section is moved to the
17241 DESCRIPTION.

17242 Issue 6

17243 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

17244 The following new requirements on POSIX implementations derive from alignment with the
17245 Single UNIX Specification:

- 17246 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
17247 required for conforming implementations of previous POSIX specifications, it was not
17248 required for UNIX applications.
- 17249 • A return value of 0 is not permitted, because {NGROUPS_MAX} cannot be 0. This is a FIPS
17250 requirement.

17251 The following changes were made to align with the IEEE P1003.1a draft standard:

- 17252 • An explanation is added that the effective group ID may be included in the supplementary
17253 group list.

17254 NAME

17255 gethostbyaddr, gethostbyname — network host database functions

17256 SYNOPSIS

17257 #include <netdb.h>

```
17258 OB        struct hostent *gethostbyaddr(const void *addr, socklen_t len,
17259              int type);
17260        struct hostent *gethostbyname(const char *name);
```

17262 DESCRIPTION

17263 These functions shall retrieve information about hosts. This information is considered to be
17264 stored in a database that can be accessed sequentially or randomly. Implementation of this
17265 database is unspecified.

17266 Note: In many cases it is implemented by the Domain Name System, as documented in RFC 1034,
17267 RFC 1035, and RFC 1886.

17268 Entries shall be returned in **hostent** structures.

17269 The *gethostbyaddr()* function shall return an entry containing addresses of address family *type* for
17270 the host with address *addr*. The *len* argument contains the length of the address pointed to by
17271 *addr*. The *gethostbyaddr()* function need not be reentrant. A function that is not required to be
17272 reentrant is not required to be thread-safe.

17273 The *gethostbyname()* function shall return an entry containing addresses of address family
17274 AF_INET for the host with name *name*. The *gethostbyname()* function need not be reentrant. A
17275 function that is not required to be reentrant is not required to be thread-safe.

17276 The *addr* argument of *gethostbyaddr()* shall be an **in_addr** structure when *type* is AF_INET. It
17277 contains a binary format (that is, not null-terminated) address in network byte order. The
17278 *gethostbyaddr()* function is not guaranteed to return addresses of address families other than
17279 AF_INET, even when such addresses exist in the database.

17280 If *gethostbyaddr()* returns successfully, then the *h_addrtype* field in the result shall be the same as
17281 the *type* argument that was passed to the function, and the *h_addr_list* field shall list a single
17282 address that is a copy of the *addr* argument that was passed to the function.

17283 The *name* argument of *gethostbyname()* shall be a node name; the behavior of *gethostbyname()*
17284 when passed a numeric address string is unspecified. For IPv4, a numeric address string shall be
17285 in the dotted-decimal notation described in *inet_addr()*.

17286 If *name* is not a numeric address string and is an alias for a valid host name, then *gethostbyname()*
17287 shall return information about the host name to which the alias refers, and *name* shall be
17288 included in the list of aliases returned.

17289 RETURN VALUE

17290 Upon successful completion, these functions shall return a pointer to a **hostent** structure if the
17291 requested entry was found, and a null pointer if the end of the database was reached or the
17292 requested entry was not found.

17293 Upon unsuccessful completion, *gethostbyaddr()* and *gethostbyname()* shall set *h_errno* to indicate
17294 the error.

17295 ERRORS

17296 These functions shall fail in the following cases. The *gethostbyaddr()* and *gethostbyname()*
17297 functions shall set *h_errno* to the value shown in the list below. Any changes to *errno* are
17298 unspecified.

17299	[HOST_NOT_FOUND]	No such host is known.
17300	[NO_DATA]	The server recognized the request and the name, but no address is available. Another type of request to the name server for the domain might return an answer.
17304	[NO_RECOVERY]	An unexpected server failure occurred which cannot be recovered.
17305	[TRY AGAIN]	A temporary and possibly transient error occurred, such as a failure of a server to respond.
17307		

17308 EXAMPLES

17309 None.

17310 APPLICATION USAGE

17311 The *gethostbyaddr()* and *gethostbyname()* functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

17313 The *getaddrinfo()* and *getnameinfo()* functions are preferred over the *gethostbyaddr()* and *gethostbyname()* functions.

17315 RATIONALE

17316 None.

17317 FUTURE DIRECTIONS

17318 The *gethostbyaddr()* and *gethostbyname()* functions may be withdrawn in a future version.

17319 SEE ALSO

17320 *endhostent()*, *endservent()*, *gai_strerror()*, *getaddrinfo()*, *h_errno*, *inet_addr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>

17322 CHANGE HISTORY

17323 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

17324 **NAME**
17325 gethostent — network host database functions
17326 **SYNOPSIS**
17327 #include <netdb.h>
17328 struct hostent *gethostent(void);
17329 **DESCRIPTION**
17330 Refer to *endhostent()*.

17331 NAME

17332 *gethostid* — get an identifier for the current host

17333 SYNOPSIS

17334 XSI `#include <unistd.h>`

17335 `long gethostid(void);`

17336

17337 DESCRIPTION

17338 The *gethostid()* function shall retrieve a 32-bit identifier for the current host.

17339 RETURN VALUE

17340 Upon successful completion, *gethostid()* shall return an identifier for the current host.

17341 ERRORS

17342 No errors are defined.

17343 EXAMPLES

17344 None.

17345 APPLICATION USAGE

17346 This volume of IEEE Std 1003.1-2001 does not define the domain in which the return value is
17347 unique.

17348 RATIONALE

17349 None.

17350 FUTURE DIRECTIONS

17351 None.

17352 SEE ALSO

17353 *random()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

17354 CHANGE HISTORY

17355 First released in Issue 4, Version 2.

17356 Issue 5

17357 Moved from X/OPEN UNIX extension to BASE.

17358 NAME

17359 *gethostname* — get name of current host

17360 SYNOPSIS

```
17361        #include <unistd.h>
17362        int gethostname(char *name, size_t namelen);
```

17363 DESCRIPTION

17364 The *gethostname()* function shall return the standard host name for the current machine. The *namelen* argument shall specify the size of the array pointed to by the *name* argument. The returned name shall be null-terminated, except that if *namelen* is an insufficient length to hold the host name, then the returned name shall be truncated and it is unspecified whether the returned name is null-terminated.

17369 Host names are limited to {HOST_NAME_MAX} bytes.

17370 RETURN VALUE

17371 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned.

17372 ERRORS

17373 No errors are defined.

17374 EXAMPLES

17375 None.

17376 APPLICATION USAGE

17377 None.

17378 RATIONALE

17379 None.

17380 FUTURE DIRECTIONS

17381 None.

17382 SEE ALSO

17383 *gethostid()*, *uname()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

17384 CHANGE HISTORY

17385 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

17386 The Open Group Base Resolution bwg2001-008 is applied, changing the *namelen* parameter from **socklen_t** to **size_t**.

17388 NAME

17389 getitimer, setitimer — get and set value of interval timer

17390 SYNOPSIS

17391 XSI #include <sys/time.h>

```
17392     int getitimer(int which, struct itimerval *value);
17393     int setitimer(int which, const struct itimerval *restrict value,
17394                     struct itimerval *restrict ovalue);
```

17395

17396 DESCRIPTION

17397 The *getitimer()* function shall store the current value of the timer specified by *which* into the
 17398 structure pointed to by *value*. The *setitimer()* function shall set the timer specified by *which* to
 17399 the value specified in the structure pointed to by *value*, and if *ovalue* is not a null pointer, store
 17400 the previous value of the timer in the structure pointed to by *ovalue*.

17401 A timer value is defined by the **itimerval** structure, specified in <sys/time.h>. If *it_value* is non-
 17402 zero, it shall indicate the time to the next timer expiration. If *it_interval* is non-zero, it shall
 17403 specify a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 shall
 17404 disable a timer, regardless of the value of *it_interval*. Setting *it_interval* to 0 shall disable a timer
 17405 after its next expiration (assuming *it_value* is non-zero).

17406 Implementations may place limitations on the granularity of timer values. For each interval
 17407 timer, if the requested timer value requires a finer granularity than the implementation supports,
 17408 the actual timer value shall be rounded up to the next supported value.

17409 An XSI-conforming implementation provides each process with at least three interval timers,
 17410 which are indicated by the *which* argument:

17411 ITIMER_REAL Decrements in real time. A SIGALRM signal is delivered when this timer
 17412 expires.

17413 ITIMER_VIRTUAL Decrements in process virtual time. It runs only when the process is
 17414 executing. A SIGVTALRM signal is delivered when it expires.

17415 ITIMER_PROF Decrements both in process virtual time and when the system is running
 17416 on behalf of the process. It is designed to be used by interpreters in
 17417 statistically profiling the execution of interpreted programs. Each time the
 17418 ITIMER_PROF timer expires, the SIGPROF signal is delivered.

17419 The interaction between *setitimer()* and any of *alarm()*, *sleep()*, or *usleep()* is unspecified.

17420 RETURN VALUE

17421 Upon successful completion, *getitimer()* or *setitimer()* shall return 0; otherwise, -1 shall be
 17422 returned and *errno* set to indicate the error.

17423 ERRORS

17424 The *setitimer()* function shall fail if:

17425 [EINVAL] The *value* argument is not in canonical form. (In canonical form, the number of
 17426 microseconds is a non-negative integer less than 1 000 000 and the number of
 17427 seconds is a non-negative integer.)

17428 The *getitimer()* and *setitimer()* functions may fail if:

17429 [EINVAL] The *which* argument is not recognized.

17430 EXAMPLES

17431 None.

17432 APPLICATION USAGE

17433 None.

17434 RATIONALE

17435 None.

17436 FUTURE DIRECTIONS

17437 None.

17438 SEE ALSO

17439 *alarm()*, *sleep()*, *timer_getoverrun()*, *ualarm()*, *usleep()*, the Base Definitions volume of
17440 IEEE Std 1003.1-2001, <signal.h>, <sys/time.h>

17441 CHANGE HISTORY

17442 First released in Issue 4, Version 2.

17443 Issue 5

17444 Moved from X/OPEN UNIX extension to BASE.

17445 Issue 6

17446 The **restrict** keyword is added to the *setitimer()* prototype for alignment with the
17447 ISO/IEC 9899:1999 standard.

17448 NAME

17449 `getlogin, getlogin_r — get login name`

17450 SYNOPSIS

17451 `#include <unistd.h>`
17452 `char *getlogin(void);`
17453 TSF `int getlogin_r(char *name, size_t namesize);`
17454

17455 DESCRIPTION

17456 The `getlogin()` function shall return a pointer to a string containing the user name associated by
17457 the login activity with the controlling terminal of the current process. If `getlogin()` returns a non-
17458 null pointer, then that pointer points to the name that the user logged in under, even if there are
17459 several login names with the same user ID.

17460 The `getlogin()` function need not be reentrant. A function that is not required to be reentrant is
17461 not required to be thread-safe.

17462 TSF The `getlogin_r()` function shall put the name associated by the login activity with the controlling
17463 terminal of the current process in the character array pointed to by `name`. The array is `namesize`
17464 characters long and should have space for the name and the terminating null character. The
17465 maximum size of the login name is `{LOGIN_NAME_MAX}`.

17466 If `getlogin_r()` is successful, `name` points to the name the user used at login, even if there are
17467 several login names with the same user ID.

17468 RETURN VALUE

17469 Upon successful completion, `getlogin()` shall return a pointer to the login name or a null pointer
17470 if the user's login name cannot be found. Otherwise, it shall return a null pointer and set `errno` to
17471 indicate the error.

17472 The return value from `getlogin()` may point to static data whose content is overwritten by each
17473 call.

17474 TSF If successful, the `getlogin_r()` function shall return zero; otherwise, an error number shall be
17475 returned to indicate the error.

17476 ERRORS

17477 The `getlogin()` and `getlogin_r()` functions may fail if:

17478 [EMFILE] `{OPEN_MAX}` file descriptors are currently open in the calling process.

17479 [ENFILE] The maximum allowable number of files is currently open in the system.

17480 [ENXIO] The calling process has no controlling terminal.

17481 The `getlogin_r()` function may fail if:

17482 TSF [ERANGE] The value of `namesize` is smaller than the length of the string to be returned
17483 including the terminating null character.

17484 EXAMPLES

17485 Getting the User Login Name

17486 The following example calls the *getlogin()* function to obtain the name of the user associated
17487 with the calling process, and passes this information to the *getpwnam()* function to get the
17488 associated user database information.

```
17489 #include <unistd.h>
17490 #include <sys/types.h>
17491 #include <pwd.h>
17492 #include <stdio.h>
17493 ...
17494 char *lgn;
17495 struct passwd *pw;
17496 ...
17497 if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
17498     fprintf(stderr, "Get of user information failed.\n"); exit(1);
17499 }
```

17500 APPLICATION USAGE

17501 Three names associated with the current process can be determined: *getpwuid(geteuid())* shall
17502 return the name associated with the effective user ID of the process; *getlogin()* shall return the
17503 name associated with the current login activity; and *getpwuid(getuid())* shall return the name
17504 associated with the real user ID of the process.

17505 The *getlogin_r()* function is thread-safe and returns values in a user-supplied buffer instead of
17506 possibly using a static data area that may be overwritten by each call.

17507 RATIONALE

17508 The *getlogin()* function returns a pointer to the user's login name. The same user ID may be
17509 shared by several login names. If it is desired to get the user database entry that is used during
17510 login, the result of *getlogin()* should be used to provide the argument to the *getpwnam()*
17511 function. (This might be used to determine the user's login shell, particularly where a single user
17512 has multiple login shells with distinct login names, but the same user ID.)

17513 The information provided by the *cuserid()* function, which was originally defined in the
17514 POSIX.1-1988 standard and subsequently removed, can be obtained by the following:

```
17515 getpwuid(geteuid())
17516 while the information provided by historical implementations of cuserid() can be obtained by:
17517 getpwuid(getuid())
```

17518 The thread-safe version of this function places the user name in a user-supplied buffer and
17519 returns a non-zero value if it fails. The non-thread-safe version may return the name in a static
17520 data area that may be overwritten by each call.

17521 FUTURE DIRECTIONS

17522 None.

17523 SEE ALSO

17524 *getpwnam()*, *getpwuid()*, *geteuid()*, *getuid()*, the Base Definitions volume of IEEE Std 1003.1-2001,
17525 *<limits.h>*, *<unistd.h>*

17526 CHANGE HISTORY

17527 First released in Issue 1. Derived from System V Release 2.0.

17528 Issue 5

17529 Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.
17530

17531 The *getlogin_r()* function is included for alignment with the POSIX Threads Extension.

17532 A note indicating that the *getlogin()* function need not be reentrant is added to the DESCRIPTION.
17533

17534 Issue 6

17535 The *getlogin_r()* function is marked as part of the Thread-Safe Functions option.

17536 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

17537 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:
17538

- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EMFILE], [ENFILE], and [ENXIO] optional error conditions are added.

17541 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
17542 its avoidance of possibly using a static data area.

17543 NAME

17544 getmsg, getpmsg — receive next message from a STREAMS file (STREAMS)

17545 SYNOPSIS

17546 XSR #include <stropts.h>

```
17547     int getmsg(int fildes, struct strbuf *restrict ctlptr,
17548                 struct strbuf *restrict dataptr, int *restrict flagsp);
17549     int getpmsg(int fildes, struct strbuf *restrict ctlptr,
17550                 struct strbuf *restrict dataptr, int *restrict bandp,
17551                 int *restrict flagsp);
```

17552

17553 DESCRIPTION

17554 The *getmsg()* function shall retrieve the contents of a message located at the head of the
 17555 STREAM head read queue associated with a STREAMS file and place the contents into one or
 17556 more buffers. The message contains either a data part, a control part, or both. The data and
 17557 control parts of the message shall be placed into separate buffers, as described below. The
 17558 semantics of each part are defined by the originator of the message.

17559 The *getpmsg()* function shall be equivalent to *getmsg()*, except that it provides finer control over
 17560 the priority of the messages received. Except where noted, all requirements on *getmsg()* also
 17561 pertain to *getpmsg()*.

17562 The *fildes* argument specifies a file descriptor referencing a STREAMS-based file.

17563 The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure, in which the *buf* member points
 17564 to a buffer in which the data or control information is to be placed, and the *maxlen* member
 17565 indicates the maximum number of bytes this buffer can hold. On return, the *len* member shall
 17566 contain the number of bytes of data or control information actually received. The *len* member
 17567 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
 17568 control information is present in the message.

17569 When *getmsg()* is called, *flagsp* should point to an integer that indicates the type of message the
 17570 process is able to receive. This is described further below.

17571 The *ctlptr* argument is used to hold the control part of the message, and *dataptr* is used to hold
 17572 the data part of the message. If *ctlptr* (or *dataptr*) is a null pointer or the *maxlen* member is -1, the
 17573 control (or data) part of the message shall not be processed and shall be left on the STREAM
 17574 head read queue, and if the *ctlptr* (or *dataptr*) is not a null pointer, *len* shall be set to -1. If the
 17575 *maxlen* member is set to 0 and there is a zero-length control (or data) part, that zero-length part
 17576 shall be removed from the read queue and *len* shall be set to 0. If the *maxlen* member is set to 0
 17577 and there are more than 0 bytes of control (or data) information, that information shall be left on
 17578 the read queue and *len* shall be set to 0. If the *maxlen* member in *ctlptr* (or *dataptr*) is less than the
 17579 control (or data) part of the message, *maxlen* bytes shall be retrieved. In this case, the remainder
 17580 of the message shall be left on the STREAM head read queue and a non-zero return value shall
 17581 be provided.

17582 By default, *getmsg()* shall process the first available message on the STREAM head read queue.
 17583 However, a process may choose to retrieve only high-priority messages by setting the integer
 17584 pointed to by *flagsp* to RS_HIPRI. In this case, *getmsg()* shall only process the next message if it is
 17585 a high-priority message. When the integer pointed to by *flagsp* is 0, any available message shall
 17586 be retrieved. In this case, on return, the integer pointed to by *flagsp* shall be set to RS_HIPRI if a
 17587 high-priority message was retrieved, or 0 otherwise.

17588 For *getpmsg()*, the flags are different. The *flagsp* argument points to a bitmask with the following
 17589 mutually-exclusive flags defined: MSG_HIPRI, MSG_BAND, and MSG_ANY. Like *getmsg()*,

17590 *getpmsg()* shall process the first available message on the STREAM head read queue. A process
17591 may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to
17592 MSG_HIPRI and the integer pointed to by *bandp* to 0. In this case, *getpmsg()* shall only process
17593 the next message if it is a high-priority message. In a similar manner, a process may choose to
17594 retrieve a message from a particular priority band by setting the integer pointed to by *flagsp* to
17595 MSG_BAND and the integer pointed to by *bandp* to the priority band of interest. In this case,
17596 *getpmsg()* shall only process the next message if it is in a priority band equal to, or greater than,
17597 the integer pointed to by *bandp*, or if it is a high-priority message. If a process wants to get the
17598 first message off the queue, the integer pointed to by *flagsp* should be set to MSG_ANY and the
17599 integer pointed to by *bandp* should be set to 0. On return, if the message retrieved was a high-
17600 priority message, the integer pointed to by *flagsp* shall be set to MSG_HIPRI and the integer
17601 pointed to by *bandp* shall be set to 0. Otherwise, the integer pointed to by *flagsp* shall be set to
17602 MSG_BAND and the integer pointed to by *bandp* shall be set to the priority band of the message.

17603 If O_NONBLOCK is not set, *getmsg()* and *getpmsg()* shall block until a message of the type
17604 specified by *flagsp* is available at the front of the STREAM head read queue. If O_NONBLOCK is
17605 set and a message of the specified type is not present at the front of the read queue, *getmsg()* and
17606 *getpmsg()* shall fail and set *errno* to [EAGAIN].

17607 If a hangup occurs on the STREAM from which messages are retrieved, *getmsg()* and *getpmsg()*
17608 shall continue to operate normally, as described above, until the STREAM head read queue is
17609 empty. Thereafter, they shall return 0 in the *len* members of *ctlptr* and *dataptr*.

17610 RETURN VALUE

17611 Upon successful completion, *getmsg()* and *getpmsg()* shall return a non-negative value. A value
17612 of 0 indicates that a full message was read successfully. A return value of MORECTL indicates
17613 that more control information is waiting for retrieval. A return value of MOREDATA indicates
17614 that more data is waiting for retrieval. A return value of the bitwise-logical OR of MORECTL
17615 and MOREDATA indicates that both types of information remain. Subsequent *getmsg()* and
17616 *getpmsg()* calls shall retrieve the remainder of the message. However, if a message of higher
17617 priority has come in on the STREAM head read queue, the next call to *getmsg()* or *getpmsg()*
17618 shall retrieve that higher-priority message before retrieving the remainder of the previous
17619 message.

17620 If the high priority control part of the message is consumed, the message shall be placed back on
17621 the queue as a normal message of band 0. Subsequent *getmsg()* and *getpmsg()* calls shall retrieve
17622 the remainder of the message. If, however, a priority message arrives or already exists on the
17623 STREAM head, the subsequent call to *getmsg()* or *getpmsg()* shall retrieve the higher-priority
17624 message before retrieving the remainder of the message that was put back.

17625 Upon failure, *getmsg()* and *getpmsg()* shall return -1 and set *errno* to indicate the error.

17626 ERRORS

17627 The *getmsg()* and *getpmsg()* functions shall fail if:

- | | |
|------------------------|--|
| 17628 [EAGAIN] | The O_NONBLOCK flag is set and no messages are available. |
| 17629 [EBADF] | The <i>fildes</i> argument is not a valid file descriptor open for reading. |
| 17630 [EBADMSG] | The queued message to be read is not valid for <i>getmsg()</i> or <i>getpmsg()</i> or a
17631 pending file descriptor is at the STREAM head. |
| 17632 [EINTR] | A signal was caught during <i>getmsg()</i> or <i>getpmsg()</i> . |
| 17633 [EINVAL] | An illegal value was specified by <i>flagsp</i> , or the STREAM or multiplexer
17634 referenced by <i>fildes</i> is linked (directly or indirectly) downstream from a
17635 multiplexer. |

17636 [ENOSTR] A STREAM is not associated with *fd*.

17637 In addition, *getmsg()* and *getpmsg()* shall fail if the STREAM head had processed an
17638 asynchronous error before the call. In this case, the value of *errno* does not reflect the result of
17639 *getmsg()* or *getpmsg()* but reflects the prior error.

17640 EXAMPLES

17641 Getting Any Message

17642 In the following example, the value of *fd* is assumed to refer to an open STREAMS file. The call
17643 to *getmsg()* retrieves any available message on the associated STREAM-head read queue,
17644 returning control and data information to the buffers pointed to by *ctrlbuf* and *databuf*,
17645 respectively.

```
17646 #include <stropts.h>
17647 ...
17648 int fd;
17649 char ctrlbuf[128];
17650 char databuf[512];
17651 struct strbuf ctrl;
17652 struct strbuf data;
17653 int flags = 0;
17654 int ret;
17655 ctrl.buf = ctrlbuf;
17656 ctrl maxlen = sizeof(ctrlbuf);
17657 data.buf = databuf;
17658 data maxlen = sizeof(databuf);
17659 ret = getmsg (fd, &ctrl, &data, &flags);
```

17660 Getting the First Message off the Queue

17661 In the following example, the call to *getpmsg()* retrieves the first available message on the
17662 associated STREAM-head read queue.

```
17663 #include <stropts.h>
17664 ...
17665 int fd;
17666 char ctrlbuf[128];
17667 char databuf[512];
17668 struct strbuf ctrl;
17669 struct strbuf data;
17670 int band = 0;
17671 int flags = MSG_ANY;
17672 int ret;
17673 ctrl.buf = ctrlbuf;
17674 ctrl maxlen = sizeof(ctrlbuf);
17675 data.buf = databuf;
17676 data maxlen = sizeof(databuf);
17677 ret = getpmsg (fd, &ctrl, &data, &band, &flags);
```

17678 APPLICATION USAGE

17679 None.

17680 RATIONALE

17681 None.

17682 FUTURE DIRECTIONS

17683 None.

17684 SEE ALSO

17685 Section 2.6 (on page 38), *poll()*, *putmsg()*, *read()*, *write()*, the Base Definitions volume of
17686 IEEE Std 1003.1-2001, <*stropts.h*>

17687 CHANGE HISTORY

17688 First released in Issue 4, Version 2.

17689 Issue 5

17690 Moved from X/OPEN UNIX extension to BASE.

17691 A paragraph regarding “high-priority control parts of messages” is added to the RETURN
17692 VALUE section.

17693 Issue 6

17694 This function is marked as part of the XSI STREAMS Option Group.

17695 The **restrict** keyword is added to the *getmsg()* and *getpmsg()* prototypes for alignment with the
17696 ISO/IEC 9899: 1999 standard.

17697 NAME

17698 getnameinfo — get name information

17699 SYNOPSIS

```
17700 #include <sys/socket.h>
17701 #include <netdb.h>
17702 int getnameinfo(const struct sockaddr *restrict sa, socklen_t salen,
17703     char *restrict node, socklen_t nodelen, char *restrict service,
17704     socklen_t servicelen, int flags);
```

1

17705 DESCRIPTION

17706 The *getnameinfo()* function shall translate a socket address to a node name and service location,
 17707 all of which are defined as in *getaddrinfo()*.

17708 The *sa* argument points to a socket address structure to be translated.

17709 IP6 If the socket address structure contains an IPv4-mapped IPv6 address or an IPv4-compatible
 17710 IPv6 address, the implementation shall extract the embedded IPv4 address and lookup the node
 17711 name for that IPv4 address.

17712 Note: The IPv6 unspecified address ("::") and the IPv6 loopback address ("::1") are not IPv4-
 17713 compatible addresses. If the address is the IPv6 unspecified address ("::"), a lookup is not
 17714 performed, and the [EAI_NONAME] error is returned.

17715 If the *node* argument is non-NULL and the *nodelen* argument is non-zero, then the *node* argument
 17716 points to a buffer able to contain up to *nodelen* characters that receives the node name as a null-
 17717 terminated string. If the *node* argument is NULL or the *nodelen* argument is zero, the node name
 17718 shall not be returned. If the node's name cannot be located, the numeric form of the address
 17719 contained in the socket address structure pointed to by the *sa* argument is returned instead of its
 17720 name.

1
1
1

17721 If the *service* argument is non-NULL and the *servicelen* argument is non-zero, then the *service*
 17722 argument points to a buffer able to contain up to *servicelen* bytes that receives the service name
 17723 as a null-terminated string. If the *service* argument is NULL or the *servicelen* argument is zero,
 17724 the service name shall not be returned. If the service's name cannot be located, the numeric form
 17725 of the service address (for example, its port number) shall be returned instead of its name.

17726 The *flags* argument is a flag that changes the default actions of the function. By default the fully-
 17727 qualified domain name (FQDN) for the host shall be returned, but:

- 17728 • If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN shall be returned
 17729 for local hosts.
- 17730 • If the flag bit NI_NUMERICHOST is set, the numeric form of the address contained in the
 17731 socket address structure pointed to by the *sa* argument shall be returned instead of its name,
 17732 under all circumstances.
- 17733 • If the flag bit NI_NAMEREQD is set, an error shall be returned if the host's name cannot be
 17734 located.
- 17735 • If the flag bit NI_NUMERICSERV is set, the numeric form of the service address shall be
 17736 returned (for example, its port number) instead of its name, under all circumstances.
- 17737 • If the flag bit NI_NUMERICSCOPE is set, the numeric form of the scope identifier shall be
 17738 returned (for example, interface index) instead of its name. This flag shall be ignored if the *sa*
 17739 argument is not an IPv6 address.
- 17740 • If the flag bit NI_DGRAM is set, this indicates that the service is a datagram service
 17741 (SOCK_DGRAM). The default behavior shall assume that the service is a stream service

1
1
1

17742 (SOCK_STREAM).

17743 **Notes:**

- 17744 1. The two NI_NUMERICxxx flags are required to support the **-n** flag that many
17745 commands provide.
- 17746 2. The NI_DGRAM flag is required for the few AF_INET and AF_INET6 port numbers (for
17747 example, [512,514]) that represent different services for UDP and TCP.

17748 The *getnameinfo()* function shall be thread-safe.

17749 **RETURN VALUE**

17750 A zero return value for *getnameinfo()* indicates successful completion; a non-zero return value
17751 indicates failure. The possible values for the failures are listed in the ERRORS section.

17752 Upon successful completion, *getnameinfo()* shall return the *node* and *service* names, if requested,
17753 in the buffers provided. The returned names are always null-terminated strings.

17754 **ERRORS**

17755 The *getnameinfo()* function shall fail and return the corresponding value if:

17756 [EAI_AGAIN] The name could not be resolved at this time. Future attempts may succeed.

17757 [EAI_BADFLAGS]

17758 The *flags* had an invalid value.

17759 [EAI_FAIL] A non-recoverable error occurred.

17760 [EAI_FAMILY] The address family was not recognized or the address length was invalid for
17761 the specified family.

17762 [EAI_MEMORY] There was a memory allocation failure.

17763 [EAI_NONAME] The name does not resolve for the supplied parameters.

17764 NI_NAMEREQD is set and the host's name cannot be located, or both
17765 *nodename* and *servname* were null.

17766 [EAI_OVERFLOW]

17767 An argument buffer overflowed. The buffer pointed to by the *node* argument
17768 or the *service* argument was too small.

17769 [EAI_SYSTEM] A system error occurred. The error code can be found in *errno*.

1
1
1

17770 **EXAMPLES**

17771 None.

17772 **APPLICATION USAGE**

17773 If the returned values are to be used as part of any further name resolution (for example, passed
17774 to *getaddrinfo()*), applications should provide buffers large enough to store any result possible
17775 on the system.

17776 Given the IPv4-mapped IPv6 address "`::ffff:1.2.3.4`", the implementation performs a
17777 lookup as if the socket address structure contains the IPv4 address "`1.2.3.4`".

17778 **RATIONALE**

17779 None.

17780 **FUTURE DIRECTIONS**

17781 None.

17782 SEE ALSO

17783 *gai_strerror()*, *getaddrinfo()*, *getservbyname()*, *inet_ntop()*, *socket()*, the Base Definitions volume of
17784 IEEE Std 1003.1-2001, <netdb.h>, <sys/socket.h>

17785 CHANGE HISTORY

17786 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

17787 The **restrict** keyword is added to the *getnameinfo()* prototype for alignment with the
17788 ISO/IEC 9899:1999 standard.

17789 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/23 is applied, making various changes in 1
17790 the SYNOPSIS and DESCRIPTION for alignment with IPv6. 1

17791 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/24 is applied, adding the 1
17792 [EAI_OVERFLOW] error to the ERRORS section. 1

17793 **NAME**17794 `getnetbyaddr`, `getnetbyname`, `getnetent` — network database functions17795 **SYNOPSIS**

```
17796     #include <netdb.h>
17797     struct netent *getnetbyaddr(uint32_t net, int type);
17798     struct netent *getnetbyname(const char *name);
17799     struct netent *getnetent(void);
```

17800 **DESCRIPTION**17801 Refer to *endnetent()*.

17802 NAME

17803 getopt, optarg, opterr, optind, optopt — command option parsing

17804 SYNOPSIS

```
17805        #include <unistd.h>
17806        int getopt(int argc, char * const argv[], const char *optstring);
17807        extern char *optarg;
17808        extern int optind, opterr, optopt;
```

17809 DESCRIPTION

17810 The *getopt()* function is a command-line parser that shall follow Utility Syntax Guidelines 3, 4, 5, 6, 7, 9, and 10 in the Base Definitions volume of IEEE Std 1003.1-2001, Section 12.2, Utility Syntax Guidelines.

17813 The parameters *argc* and *argv* are the argument count and argument array as passed to *main()* (see *exec*). The argument *optstring* is a string of recognized option characters; if a character is followed by a colon, the option takes an argument. All option characters allowed by Utility Syntax Guideline 3 are allowed in *optstring*. The implementation may accept other characters as an extension.

17818 The variable *optind* is the index of the next element of the *argv[]* vector to be processed. It shall be initialized to 1 by the system, and *getopt()* shall update it when it finishes with each element of *argv[]*. When an element of *argv[]* contains multiple option characters, it is unspecified how *getopt()* determines which options have already been processed.

17822 The *getopt()* function shall return the next option character (if one is found) from *argv* that matches a character in *optstring*, if there is one that matches. If the option takes an argument, *getopt()* shall set the variable *optarg* to point to the option-argument as follows:

1. If the option was the last character in the string pointed to by an element of *argv*, then *optarg* shall contain the next element of *argv*, and *optind* shall be incremented by 2. If the resulting value of *optind* is greater than *argc*, this indicates a missing option-argument, and *getopt()* shall return an error indication.
2. Otherwise, *optarg* shall point to the string following the option character in that element of *argv*, and *optind* shall be incremented by 1.

17831 If, when *getopt()* is called:

17832 *argv[optind]* is a null pointer
17833 **argv[optind]* is not the character –
17834 *argv[optind]* points to the string “–”

17835 *getopt()* shall return -1 without changing *optind*. If:

17836 *argv[optind]* points to the string “––”

17837 *getopt()* shall return -1 after incrementing *optind*.

17838 If *getopt()* encounters an option character that is not contained in *optstring*, it shall return the question-mark ('?') character. If it detects a missing option-argument, it shall return the colon character (':') if the first character of *optstring* was a colon, or a question-mark character ('?') otherwise. In either case, *getopt()* shall set the variable *optopt* to the option character that caused the error. If the application has not set the variable *opterr* to 0 and the first character of *optstring* is not a colon, *getopt()* shall also print a diagnostic message to *stderr* in the format specified for the *getopts* utility.

17845 The *getopt()* function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

17847 RETURN VALUE

17848 The *getopt()* function shall return the next option character specified on the command line.

17849 A colon (':') shall be returned if *getopt()* detects a missing argument and the first character of
17850 *optstring* was a colon (':').

17851 A question mark ('?') shall be returned if *getopt()* encounters an option character not in
17852 *optstring* or detects a missing argument and the first character of *optstring* was not a colon (':').

17853 Otherwise, *getopt()* shall return -1 when all command line options are parsed.

17854 ERRORS

17855 No errors are defined.

17856 EXAMPLES

17857 Parsing Command Line Options

17858 The following code fragment shows how you might process the arguments for a utility that can
17859 take the mutually-exclusive options *a* and *b* and the options *f* and *o*, both of which require
17860 arguments:

```
17861 #include <unistd.h>
17862 int
17863 main(int argc, char *argv[ ])
17864 {
17865     int c;
17866     int bflg, aflg, errflg;
17867     char *ifile;
17868     char *ofile;
17869     extern char * optarg;
17870     extern int optind, getopt;
17871     . .
17872     while ((c = getopt(argc, argv, ":abf:o:")) != -1) {
17873         switch(c) {
17874             case 'a':
17875                 if (bflg)
17876                     errflg++;
17877                 else
17878                     aflg++;
17879                 break;
17880             case 'b':
17881                 if (aflg)
17882                     errflg++;
17883                 else {
17884                     bflg++;
17885                     bproc();
17886                 }
17887                 break;
17888             case 'f':
17889                 ifile = optarg;
17890                 break;
17891             case 'o':
17892                 ofile = optarg;
17893                 break;
```

```

17894         case '':           /* -f or -o without operand */
17895             fprintf(stderr,
17896                     "Option -%c requires an operand\n", optopt);
17897             errflg++;
17898             break;
17899         case '?':
17900             fprintf(stderr,
17901                     "Unrecognized option: -%c\n", optopt);
17902             errflg++;
17903         }
17904     }
17905     if (errflg) {
17906         fprintf(stderr, "usage: . . . ");
17907         exit(2);
17908     }
17909     for ( ; optind < argc; optind++) {
17910         if (access(argv[optind], R_OK)) {
17911             . . .
17912         }

```

17913 This code accepts any of the following as equivalent:

```

17914 cmd -ao arg path path
17915 cmd -a -o arg path path
17916 cmd -o arg -a path path
17917 cmd -a -o arg -- path path
17918 cmd -a -oarg path path
17919 cmd -aoarg path path

```

17920 Checking Options and Arguments

17921 The following example parses a set of command line options and prints messages to standard
17922 output for each option and argument that it encounters.

```

17923 #include <unistd.h>
17924 #include <stdio.h>
17925 ...
17926 int c;
17927 char *filename;
17928 extern char *optarg;
17929 extern int optind, optopt, opterr;
17930 ...
17931 while ((c = getopt(argc, argv, ":abf:")) != -1) {
17932     switch(c) {
17933         case 'a':
17934             printf("a is set\n");
17935             break;
17936         case 'b':
17937             printf("b is set\n");
17938             break;
17939         case 'f':
17940             filename = optarg;
17941             printf("filename is %s\n", filename);
17942             break;

```

```
17943     case :::
17944         printf("-%c without filename\n", optopt);
17945         break;
17946     case '?':
17947         printf("unknown arg %c\n", optopt);
17948         break;
17949     }
17950 }
```

17951 Selecting Options from the Command Line

17952 The following example selects the type of database routines the user wants to use based on the
17953 *Options* argument.

```
17954 #include <unistd.h>
17955 #include <string.h>
17956 ...
17957 char *Options = "hdbtl";
17958 ...
17959 int dbtype, i;
17960 char c;
17961 char *st;
17962 ...
17963 dbtype = 0;
17964 while ((c = getopt(argc, argv, Options)) != -1) {
17965     if ((st = strchr(Options, c)) != NULL) {
17966         dbtype = st - Options;
17967         break;
17968     }
17969 }
```

17970 APPLICATION USAGE

17971 The *getopt()* function is only required to support option characters included in Utility Syntax
17972 Guideline 3. Many historical implementations of *getopt()* support other characters as options.
17973 This is an allowed extension, but applications that use extensions are not maximally portable.
17974 Note that support for multi-byte option characters is only possible when such characters can be
17975 represented as type **int**.

17976 RATIONALE

17977 The *optopt* variable represents historical practice and allows the application to obtain the identity
17978 of the invalid option.

17979 The description has been written to make it clear that *getopt()*, like the *getopts* utility, deals with
17980 option-arguments whether separated from the option by <blank>s or not. Note that the
17981 requirements on *getopt()* and *getopts* are more stringent than the Utility Syntax Guidelines.

17982 The *getopt()* function shall return -1, rather than EOF, so that <**stdio.h**> is not required.

17983 The special significance of a colon as the first character of *optstring* makes *getopt()* consistent
17984 with the *getopts* utility. It allows an application to make a distinction between a missing
17985 argument and an incorrect option letter without having to examine the option letter. It is true
17986 that a missing argument can only be detected in one case, but that is a case that has to be
17987 considered.

17988 FUTURE DIRECTIONS

17989 None.

17990 SEE ALSO

17991 *exec*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>, the Shell and Utilities
17992 volume of IEEE Std 1003.1-2001

17993 CHANGE HISTORY

17994 First released in Issue 1. Derived from Issue 1 of the SVID.

17995 Issue 5

17996 A note indicating that the *getopt()* function need not be reentrant is added to the DESCRIPTION.

17997 Issue 6

17998 IEEE PASC Interpretation 1003.2 #150 is applied.

17999 NAME

18000 getpeername — get the name of the peer socket

18001 SYNOPSIS

```
18002        #include <sys/socket.h>
18003        int getpeername(int socket, struct sockaddr *restrict address,
18004              socklen_t *restrict address_len);
```

18005 DESCRIPTION

18006 The *getpeername()* function shall retrieve the peer address of the specified socket, store this
18007 address in the **sockaddr** structure pointed to by the *address* argument, and store the length of this
18008 address in the object pointed to by the *address_len* argument.

18009 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
18010 the stored address shall be truncated.

18011 If the protocol permits connections by unbound clients, and the peer is not bound, then the value
18012 stored in the object pointed to by *address* is unspecified.

18013 RETURN VALUE

18014 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
18015 indicate the error.

18016 ERRORS

18017 The *getpeername()* function shall fail if:

18018 [EBADF] The *socket* argument is not a valid file descriptor.

18019 [EINVAL] The socket has been shut down.

18020 [ENOTCONN] The socket is not connected or otherwise has not had the peer pre-specified.

18021 [ENOTSOCK] The *socket* argument does not refer to a socket.

18022 [EOPNOTSUPP] The operation is not supported for the socket protocol.

18023 The *getpeername()* function may fail if:

18024 [ENOBUFS] Insufficient resources were available in the system to complete the call.

18025 EXAMPLES

18026 None.

18027 APPLICATION USAGE

18028 None.

18029 RATIONALE

18030 None.

18031 FUTURE DIRECTIONS

18032 None.

18033 SEE ALSO

18034 *accept()*, *bind()*, *getsockname()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001,
18035 <sys/socket.h>

18036 CHANGE HISTORY

18037 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

18038 The **restrict** keyword is added to the *getpeername()* prototype for alignment with the
18039 ISO/IEC 9899:1999 standard.

18040 NAME

18041 *getpgid* — get the process group ID for a process

18042 SYNOPSIS

18043 XSI #include <unistd.h>

18044 pid_t getpgid(pid_t *pid*);

18045

18046 DESCRIPTION

18047 The *getpgid*() function shall return the process group ID of the process whose process ID is equal to *pid*. If *pid* is equal to 0, *getpgid*() shall return the process group ID of the calling process.

18049 RETURN VALUE

18050 Upon successful completion, *getpgid*() shall return a process group ID. Otherwise, it shall return (pid_t)-1 and set *errno* to indicate the error.

18052 ERRORS

18053 The *getpgid*() function shall fail if:

18054 [EPERM] The process whose process ID is equal to *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of that process from the calling process.

18057 [ESRCH] There is no process with a process ID equal to *pid*.

18058 The *getpgid*() function may fail if:

18059 [EINVAL] The value of the *pid* argument is invalid.

18060 EXAMPLES

18061 None.

18062 APPLICATION USAGE

18063 None.

18064 RATIONALE

18065 None.

18066 FUTURE DIRECTIONS

18067 None.

18068 SEE ALSO

18069 *exec*, *fork*(), *getpgrp*(), *getpid*(), *getsid*(), *setpgid*(), *setsid*(), the Base Definitions volume of
18070 IEEE Std 1003.1-2001, <unistd.h>

18071 CHANGE HISTORY

18072 First released in Issue 4, Version 2.

18073 Issue 5

18074 Moved from X/OPEN UNIX extension to BASE.

18075 NAME

18076 *getpgrp* — get the process group ID of the calling process

18077 SYNOPSIS

```
18078        #include <unistd.h>
18079        pid_t getpgrp(void);
```

18080 DESCRIPTION

18081 The *getpgrp()* function shall return the process group ID of the calling process.

18082 RETURN VALUE

18083 The *getpgrp()* function shall always be successful and no return value is reserved to indicate an error.

18085 ERRORS

18086 No errors are defined.

18087 EXAMPLES

18088 None.

18089 APPLICATION USAGE

18090 None.

18091 RATIONALE

18092 4.3 BSD provides a *getpgrp()* function that returns the process group ID for a specified process.
18093 Although this function supports job control, all known job control shells always specify the
18094 calling process with this function. Thus, the simpler System V *getpgrp()* suffices, and the added
18095 complexity of the 4.3 BSD *getpgrp()* is provided by the XSI extension *getpgid()*.

18096 FUTURE DIRECTIONS

18097 None.

18098 SEE ALSO

18099 *exec*, *fork()*, *getpgid()*, *getpid()*, *getppid()*, *kill()*, *setpgid()*, *setsid()*, the Base Definitions volume of
18100 IEEE Std 1003.1-2001, <*sys/types.h*>, <*unistd.h*>

18101 CHANGE HISTORY

18102 First released in Issue 1. Derived from Issue 1 of the SVID.

18103 Issue 6

18104 In the SYNOPSIS, the optional include of the <*sys/types.h*> header is removed.

18105 The following new requirements on POSIX implementations derive from alignment with the
18106 Single UNIX Specification:

- 18107 • The requirement to include <*sys/types.h*> has been removed. Although <*sys/types.h*> was
18108 required for conforming implementations of previous POSIX specifications, it was not
18109 required for UNIX applications.

18110 NAME

18111 `getpid` — get the process ID

18112 SYNOPSIS

```
18113        #include <unistd.h>
18114        pid_t getpid(void);
```

18115 DESCRIPTION

18116 The `getpid()` function shall return the process ID of the calling process.

18117 RETURN VALUE

18118 The `getpid()` function shall always be successful and no return value is reserved to indicate an error.

18120 ERRORS

18121 No errors are defined.

18122 EXAMPLES

18123 None.

18124 APPLICATION USAGE

18125 None.

18126 RATIONALE

18127 None.

18128 FUTURE DIRECTIONS

18129 None.

18130 SEE ALSO

18131 `exec`, `fork()`, `getpgrp()`, `getppid()`, `kill()`, `setpgid()`, `setsid()`, the Base Definitions volume of
18132 IEEE Std 1003.1-2001, `<sys/types.h>`, `<unistd.h>`

18133 CHANGE HISTORY

18134 First released in Issue 1. Derived from Issue 1 of the SVID.

18135 Issue 6

18136 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

18137 The following new requirements on POSIX implementations derive from alignment with the
18138 Single UNIX Specification:

- 18139 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
18140 required for conforming implementations of previous POSIX specifications, it was not
18141 required for UNIX applications.

18142 NAME

18143 **getpmsg** — receive next message from a STREAMS file

18144 SYNOPSIS

18145 XSI #include <stropts.h>

```
18146        int getpmsg(int fildes, struct strbuf *restrict ctlptr,
18147            struct strbuf *restrict dataptr, int *restrict bandp,
18148            int *restrict flagsp);
```

18149

18150 DESCRIPTION

18151 Refer to *getmsg()*.

18152 NAME

18153 `getppid` — get the parent process ID

18154 SYNOPSIS

```
18155        #include <unistd.h>
18156        pid_t getppid(void);
```

18157 DESCRIPTION

18158 The `getppid()` function shall return the parent process ID of the calling process.

18159 RETURN VALUE

18160 The `getppid()` function shall always be successful and no return value is reserved to indicate an error.

18162 ERRORS

18163 No errors are defined.

18164 EXAMPLES

18165 None.

18166 APPLICATION USAGE

18167 None.

18168 RATIONALE

18169 None.

18170 FUTURE DIRECTIONS

18171 None.

18172 SEE ALSO

18173 `exec`, `fork()`, `getpgid()`, `getpgrp()`, `getpid()`, `kill()`, `setpgid()`, `setsid()`, the Base Definitions volume of
18174 IEEE Std 1003.1-2001, `<sys/types.h>`, `<unistd.h>`

18175 CHANGE HISTORY

18176 First released in Issue 1. Derived from Issue 1 of the SVID.

18177 Issue 6

18178 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

18179 The following new requirements on POSIX implementations derive from alignment with the
18180 Single UNIX Specification:

- 18181 • The requirement to include `<sys/types.h>` has been removed. Although `<sys/types.h>` was
18182 required for conforming implementations of previous POSIX specifications, it was not
18183 required for UNIX applications.

18184 NAME

18185 *getpriority*, *setpriority* — get and set the nice value

18186 SYNOPSIS

18187 XSI #include <sys/resource.h>

```
18188        int getpriority(int which, id_t who);
18189        int setpriority(int which, id_t who, int value);
```

18190

18191 DESCRIPTION

18192 The *getpriority*() function shall obtain the nice value of a process, process group, or user. The
 18193 *setpriority*() function shall set the nice value of a process, process group, or user to
 18194 *value*+{NZERO}.

18195 Target processes are specified by the values of the *which* and *who* arguments. The *which*
 18196 argument may be one of the following values: PRIO_PROCESS, PRIO_PGRP, or PRIO_USER,
 18197 indicating that the *who* argument is to be interpreted as a process ID, a process group ID, or an
 18198 effective user ID, respectively. A 0 value for the *who* argument specifies the current process,
 18199 process group, or user.

18200 The nice value set with *setpriority*() shall be applied to the process. If the process is multi-
 18201 threaded, the nice value shall affect all system scope threads in the process.

18202 If more than one process is specified, *getpriority*() shall return value {NZERO} less than the
 18203 lowest nice value pertaining to any of the specified processes, and *setpriority*() shall set the nice
 18204 values of all of the specified processes to *value*+{NZERO}.

18205 The default nice value is {NZERO}; lower nice values shall cause more favorable scheduling.
 18206 While the range of valid nice values is [0,{NZERO}*2-1], implementations may enforce more
 18207 restrictive limits. If *value*+{NZERO} is less than the system's lowest supported nice value,
 18208 *setpriority*() shall set the nice value to the lowest supported value; if *value*+{NZERO} is greater
 18209 than the system's highest supported nice value, *setpriority*() shall set the nice value to the highest
 18210 supported value.

18211 Only a process with appropriate privileges can lower its nice value.

18212 PS|TPS Any processes or threads using SCHED_FIFO or SCHED_RR shall be unaffected by a call to
 18213 *setpriority*(). This is not considered an error. A process which subsequently reverts to
 18214 SCHED_OTHER need not have its priority affected by such a *setpriority*() call.

18215 The effect of changing the nice value may vary depending on the process-scheduling algorithm
 18216 in effect.

18217 Since *getpriority*() can return the value -1 on successful completion, it is necessary to set *errno* to
 18218 0 prior to a call to *getpriority*(). If *getpriority*() returns the value -1, then *errno* can be checked to
 18219 see if an error occurred or if the value is a legitimate nice value.

18220 RETURN VALUE

18221 Upon successful completion, *getpriority*() shall return an integer in the range -{NZERO} to
 18222 {NZERO}-1. Otherwise, -1 shall be returned and *errno* set to indicate the error.

18223 Upon successful completion, *setpriority*() shall return 0; otherwise, -1 shall be returned and *errno*
 18224 set to indicate the error.

18225 ERRORS

18226 The *getpriority*() and *setpriority*() functions shall fail if:

18227 [ESRCH]	No process could be located using the <i>which</i> and <i>who</i> argument values 18228 specified.
----------------------	--

- 18229 [EINVAL] The value of the *which* argument was not recognized, or the value of the *who* argument is not a valid process ID, process group ID, or user ID.
- 18231 In addition, *setpriority()* may fail if:
- 18232 [EPERM] A process was located, but neither the real nor effective user ID of the executing process match the effective user ID of the process whose nice value is being changed.
- 18235 [EACCES] A request was made to change the nice value to a lower numeric value and the current process does not have appropriate privileges.

18237 EXAMPLES

18238 Using getpriority()

18239 The following example returns the current scheduling priority for the process ID returned by the
18240 call to *getpid()*.

```
18241 #include <sys/resource.h>
18242 ...
18243 int which = PRIO_PROCESS;
18244 id_t pid;
18245 int ret;
18246
18247 pid = getpid();
ret = getpriority(which, pid);
```

18248 Using setpriority()

18249 The following example sets the priority for the current process ID to -20.

```
18250 #include <sys/resource.h>
18251 ...
18252 int which = PRIO_PROCESS;
18253 id_t pid;
18254 int priority = -20;
18255 int ret;
18256
18257 pid = getpid();
ret = setpriority(which, pid, priority);
```

18258 APPLICATION USAGE

18259 The *getpriority()* and *setpriority()* functions work with an offset nice value (nice value
18260 $-\{NZERO\}$). The nice value is in the range $[0,2^*\{NZERO\} - 1]$, while the return value for
18261 *getpriority()* and the third parameter for *setpriority()* are in the range $[-\{NZERO\},\{NZERO\} - 1]$.

18262 RATIONALE

18263 None.

18264 FUTURE DIRECTIONS

18265 None.

18266 SEE ALSO

18267 *nice()*, *sched_get_priority_max()*, *sched_setscheduler()*, the Base Definitions volume of
18268 IEEE Std 1003.1-2001, <sys/resource.h>

18269 CHANGE HISTORY

18270 First released in Issue 4, Version 2.

18271 Issue 5

18272 Moved from X/OPEN UNIX extension to BASE.

18273 The DESCRIPTION is reworded in terms of the nice value rather than *priority* to avoid confusion
18274 with functionality in the POSIX Realtime Extension.

18275 **NAME**

18276 getprotobynumber, getprotobynumber, getprotoent — network protocol database functions

18277 **SYNOPSIS**

```
18278     #include <netdb.h>
18279
18280     struct protoent *getprotobynumber(const char *name);
18281     struct protoent *getprotobynumber(int proto);
18282     struct protoent *getprotoent(void);
```

18282 **DESCRIPTION**18283 Refer to *endprotoent()*.

18284 **NAME**18285 `getpwent` — get user database entry18286 **SYNOPSIS**18287 XSI `#include <pwd.h>`18288 `struct passwd *getpwent(void);`

18289

18290 **DESCRIPTION**18291 Refer to *endpwent()*.

18292 NAME

18293 `getpwnam`, `getpwnam_r` — search user database for a name

18294 SYNOPSIS

```
18295        #include <pwd.h>
18296
18297        struct passwd *getpwnam(const char *name);
18298        int getpwnam_r(const char *name, struct passwd *pwd, char *buffer,
18299                      size_t bufsize, struct passwd **result);
```

18300 DESCRIPTION

18301 The `getpwnam()` function shall search the user database for an entry with a matching *name*.

18302 The `getpwnam()` function need not be reentrant. A function that is not required to be reentrant is
18303 not required to be thread-safe.

18304 Applications wishing to check for error situations should set *errno* to 0 before calling
18305 `getpwnam()`. If `getpwnam()` returns a null pointer and *errno* is non-zero, an error occurred.

18306 TSF The `getpwnam_r()` function shall update the **passwd** structure pointed to by *pwd* and store a
18307 pointer to that structure at the location pointed to by *result*. The structure shall contain an entry
18308 from the user database with a matching *name*. Storage referenced by the structure is allocated
18309 from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
18310 maximum size needed for this buffer can be determined with the `{_SC_GETPW_R_SIZE_MAX}`
18311 `sysconf()` parameter. A NULL pointer shall be returned at the location pointed to by *result* on
18312 error or if the requested entry is not found.

18313 RETURN VALUE

18314 The `getpwnam()` function shall return a pointer to a **struct passwd** with the structure as defined
18315 in `<pwd.h>` with a matching entry if found. A null pointer shall be returned if the requested
18316 entry is not found, or an error occurs. On error, *errno* shall be set to indicate the error.

18317 The return value may point to a static area which is overwritten by a subsequent call to
18318 `getpwent()`, `getpwnam()`, or `getpwuid()`.

18319 TSF If successful, the `getpwnam_r()` function shall return zero; otherwise, an error number shall be
18320 returned to indicate the error.

18321 ERRORS

18322 The `getpwnam()` and `getpwnam_r()` functions may fail if:

18323 [EIO]	An I/O error has occurred.
18324 [EINTR]	A signal was caught during <code>getpwnam()</code> .
18325 [EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
18326 [ENFILE]	The maximum allowable number of files is currently open in the system.
18327	The <code>getpwnam_r()</code> function may fail if:
18328 TSF [ERANGE]	Insufficient storage was supplied via <i>buffer</i> and <i>bufsize</i> to contain the data to 18329 be referenced by the resulting passwd structure.

18330 EXAMPLES**18331 Getting an Entry for the Login Name**

18332 The following example uses the *getlogin()* function to return the name of the user who logged in;
18333 this information is passed to the *getpwnam()* function to get the user database entry for that user.

```
18334 #include <sys/types.h>
18335 #include <pwd.h>
18336 #include <unistd.h>
18337 #include <stdio.h>
18338 #include <stdlib.h>
18339 ...
18340 char *lgn;
18341 struct passwd *pw;
18342 ...
18343 if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
18344     fprintf(stderr, "Get of user information failed.\n"); exit(1);
18345 }
18346 ...
```

18347 APPLICATION USAGE

18348 Three names associated with the current process can be determined: *getpwuid(geteuid())* returns
18349 the name associated with the effective user ID of the process; *getlogin()* returns the name
18350 associated with the current login activity; and *getpwuid(getuid())* returns the name associated
18351 with the real user ID of the process.

18352 The *getpwnam_r()* function is thread-safe and returns values in a user-supplied buffer instead of
18353 possibly using a static data area that may be overwritten by each call.

18354 RATIONALE

18355 None.

18356 FUTURE DIRECTIONS

18357 None.

18358 SEE ALSO

18359 *getpwuid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <pwd.h>,
18360 <sys/types.h>

18361 CHANGE HISTORY

18362 First released in Issue 1. Derived from System V Release 2.0.

18363 Issue 5

18364 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
18365 VALUE section.

18366 The *getpwnam_r()* function is included for alignment with the POSIX Threads Extension.

18367 A note indicating that the *getpwnam()* function need not be reentrant is added to the
18368 DESCRIPTION.

18369 Issue 6

18370 The *getpwnam_r()* function is marked as part of the Thread-Safe Functions option.

18371 The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION
18372 describing matching the *name*.

18373 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

18374 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

18375 The following new requirements on POSIX implementations derive from alignment with the
18376 Single UNIX Specification:

- 18377 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
18378 required for conforming implementations of previous POSIX specifications, it was not
18379 required for UNIX applications.
- 18380 • In the RETURN VALUE section, the requirement to set *errno* on error is added.
- 18381 • The [EMFILE], [ENFILE], and [ENXIO] optional error conditions are added.

18382 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
18383 its avoidance of possibly using a static data area.

18384 IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the
18385 buffer from *bufsize* characters to bytes.

18386 NAME

18387 `getpwuid`, `getpwuid_r` — search user database for a user ID

18388 SYNOPSIS

```
18389        #include <pwd.h>
18390
18391        struct passwd *getpwuid(uid_t uid);
18392        TSF      int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,
18393                    size_t bufsize, struct passwd **result);
```

18394 DESCRIPTION

18395 The `getpwuid()` function shall search the user database for an entry with a matching *uid*.

18396 The `getpwuid()` function need not be reentrant. A function that is not required to be reentrant is
18397 not required to be thread-safe.

18398 Applications wishing to check for error situations should set *errno* to 0 before calling `getpwuid()`.
18399 If `getpwuid()` returns a null pointer and *errno* is set to non-zero, an error occurred.

18400 TSF The `getpwuid_r()` function shall update the **passwd** structure pointed to by *pwd* and store a
18401 pointer to that structure at the location pointed to by *result*. The structure shall contain an entry
18402 from the user database with a matching *uid*. Storage referenced by the structure is allocated
18403 from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The
18404 maximum size needed for this buffer can be determined with the `{_SC_GETPW_R_SIZE_MAX}`
18405 `sysconf()` parameter. A NULL pointer shall be returned at the location pointed to by *result* on
18406 error or if the requested entry is not found.

18407 RETURN VALUE

18408 The `getpwuid()` function shall return a pointer to a **struct passwd** with the structure as defined in
18409 `<pwd.h>` with a matching entry if found. A null pointer shall be returned if the requested entry
18410 is not found, or an error occurs. On error, *errno* shall be set to indicate the error.

18411 The return value may point to a static area which is overwritten by a subsequent call to
18412 `getpwent()`, `getpwnam()`, or `getpwuid()`.

18413 TSF If successful, the `getpwuid_r()` function shall return zero; otherwise, an error number shall be
18414 returned to indicate the error.

18415 ERRORS

18416 The `getpwuid()` and `getpwuid_r()` functions may fail if:

18417 [EIO] An I/O error has occurred.
18418 [EINTR] A signal was caught during `getpwuid()`.
18419 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.
18420 [ENFILE] The maximum allowable number of files is currently open in the system.

18421 The `getpwuid_r()` function may fail if:

18422 TSF [ERANGE] Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to
18423 be referenced by the resulting **passwd** structure.

18424 EXAMPLES

18425 **Getting an Entry for the Root User**

18426 The following example gets the user database entry for the user with user ID 0 (root).

```
18427 #include <sys/types.h>
18428 #include <pwd.h>
18429 ...
18430 uid_t id = 0;
18431 struct passwd *pwd;
18432 pwd = getpwuid(id);
```

18433 **Finding the Name for the Effective User ID**

18434 The following example defines *pws* as a pointer to a structure of type **passwd**, which is used to
18435 store the structure pointer returned by the call to the *getpwuid()* function. The *geteuid()* function
18436 shall return the effective user ID of the calling process; this is used as the search criteria for the
18437 *getpwuid()* function. The call to *getpwuid()* shall return a pointer to the structure containing that
18438 user ID value.

```
18439 #include <unistd.h>
18440 #include <sys/types.h>
18441 #include <pwd.h>
18442 ...
18443 struct passwd *pws;
18444 pws = getpwuid(geteuid());
```

18445 **Finding an Entry in the User Database**

18446 The following example uses *getpwuid()* to search the user database for a user ID that was
18447 previously stored in a **stat** structure, then prints out the user name if it is found. If the user is not
18448 found, the program prints the numeric value of the user ID for the entry.

```
18449 #include <sys/types.h>
18450 #include <pwd.h>
18451 #include <stdio.h>
18452 ...
18453 struct stat statbuf;
18454 struct passwd *pwd;
18455 ...
18456 if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
18457     printf(" %-8.8s", pwd->pw_name);
18458 else
18459     printf(" %-8d", statbuf.st_uid);
```

18460 **APPLICATION USAGE**

18461 Three names associated with the current process can be determined: *getpwuid(geteuid())* returns
18462 the name associated with the effective user ID of the process; *getlogin()* returns the name
18463 associated with the current login activity; and *getpwuid(getuid())* returns the name associated
18464 with the real user ID of the process.

18465 The *getpwuid_r()* function is thread-safe and returns values in a user-supplied buffer instead of
18466 possibly using a static data area that may be overwritten by each call.

18467 **RATIONALE**

18468 None.

18469 **FUTURE DIRECTIONS**

18470 None.

18471 **SEE ALSO**18472 *getpwnam()*, *geteuid()*, *getuid()*, *getlogin()*, the Base Definitions volume of IEEE Std 1003.1-2001,
18473 <limits.h>, <pwd.h>, <sys/types.h>18474 **CHANGE HISTORY**

18475 First released in Issue 1. Derived from System V Release 2.0.

18476 **Issue 5**18477 Normative text previously in the APPLICATION USAGE section is moved to the RETURN
18478 VALUE section.18479 The *getpwuid_r()* function is included for alignment with the POSIX Threads Extension.18480 A note indicating that the *getpwuid()* function need not be reentrant is added to the
18481 DESCRIPTION.18482 **Issue 6**18483 The *getpwuid_r()* function is marked as part of the Thread-Safe Functions option.18484 The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION
18485 describing matching the *uid*.

18486 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

18487 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

18488 The following new requirements on POSIX implementations derive from alignment with the
18489 Single UNIX Specification:

- 18490 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
18491 required for conforming implementations of previous POSIX specifications, it was not
18492 required for UNIX applications.

- 18493 • In the RETURN VALUE section, the requirement to set *errno* on error is added.

- 18494 • The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

18495 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
18496 its avoidance of possibly using a static data area.18497 IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the
18498 buffer from *bufsize* characters to bytes.

18499 NAME

18500 getrlimit, setrlimit — control maximum resource consumption

18501 SYNOPSIS

18502 XSI #include <sys/resource.h>

18503 int getrlimit(int resource, struct rlimit *rlp);

18504 int setrlimit(int resource, const struct rlimit *rlp);

18505

18506 DESCRIPTION

18507 The *getrlimit()* function shall get, and the *setrlimit()* function shall set, limits on the consumption
18508 of a variety of resources.18509 Each call to either *getrlimit()* or *setrlimit()* identifies a specific resource to be operated upon as
18510 well as a resource limit. A resource limit is represented by an **rlimit** structure. The *rlim_cur*
18511 member specifies the current or soft limit and the *rlim_max* member specifies the maximum or
18512 hard limit. Soft limits may be changed by a process to any value that is less than or equal to the
18513 hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or
18514 equal to the soft limit. Only a process with appropriate privileges can raise a hard limit. Both
18515 hard and soft limits can be changed in a single call to *setrlimit()* subject to the constraints
18516 described above.18517 The value **RLIM_INFINITY**, defined in <sys/resource.h>, shall be considered to be larger than
18518 any other limit value. If a call to *getrlimit()* returns **RLIM_INFINITY** for a resource, it means the
18519 implementation shall not enforce limits on that resource. Specifying **RLIM_INFINITY** as any
18520 resource limit value on a successful call to *setrlimit()* shall inhibit enforcement of that resource
18521 limit.

18522 The following resources are defined:

18523 **RLIMIT_CORE** This is the maximum size of a **core** file, in bytes, that may be created by a
18524 process. A limit of 0 shall prevent the creation of a **core** file. If this limit is
18525 exceeded, the writing of a **core** file shall terminate at this size.18526 **RLIMIT_CPU** This is the maximum amount of CPU time, in seconds, used by a process.
18527 If this limit is exceeded, SIGXCPU shall be generated for the process. If
18528 the process is catching or ignoring SIGXCPU, or all threads belonging to
18529 that process are blocking SIGXCPU, the behavior is unspecified.18530 **RLIMIT_DATA** This is the maximum size of a process' data segment, in bytes. If this limit
18531 is exceeded, the *malloc()* function shall fail with *errno* set to [ENOMEM].18532 **RLIMIT_FSIZE** This is the maximum size of a file, in bytes, that may be created by a
18533 process. If a write or truncate operation would cause this limit to be
18534 exceeded, SIGXFSZ shall be generated for the thread. If the thread is
18535 blocking, or the process is catching or ignoring SIGXFSZ, continued
18536 attempts to increase the size of a file from end-of-file to beyond the limit
18537 shall fail with *errno* set to [EFBIG].18538 **RLIMIT_NOFILE** This is a number one greater than the maximum value that the system
18539 may assign to a newly-created descriptor. If this limit is exceeded,
18540 functions that allocate a file descriptor shall fail with *errno* set to
18541 [EMFILE]. This limit constrains the number of file descriptors that a
18542 process may allocate.18543 **RLIMIT_STACK** This is the maximum size of the initial thread's stack, in bytes. The
18544 implementation does not automatically grow the stack beyond this limit. 2

18545 If this limit is exceeded, SIGSEGV shall be generated for the thread. If the
18546 thread is blocking SIGSEGV, or the process is ignoring or catching
18547 SIGSEGV and has not made arrangements to use an alternate stack, the
18548 disposition of SIGSEGV shall be set to SIG_DFL before it is generated.

18549 RLIMIT_AS This is the maximum size of a process' total available memory, in bytes. If
18550 this limit is exceeded, the *malloc()* and *mmap()* functions shall fail with
18551 *errno* set to [ENOMEM]. In addition, the automatic stack growth fails
18552 with the effects outlined above.

18553 When using the *getrlimit()* function, if a resource limit can be represented correctly in an object
18554 of type **rlim_t**, then its representation is returned; otherwise, if the value of the resource limit is
18555 equal to that of the corresponding saved hard limit, the value returned shall be
18556 RLIM_SAVED_MAX; otherwise, the value returned shall be RLIM_SAVED_CUR.

18557 When using the *setrlimit()* function, if the requested new limit is RLIM_INFINITY, the new limit
18558 shall be "no limit"; otherwise, if the requested new limit is RLIM_SAVED_MAX, the new limit
18559 shall be the corresponding saved hard limit; otherwise, if the requested new limit is
18560 RLIM_SAVED_CUR, the new limit shall be the corresponding saved soft limit; otherwise, the
18561 new limit shall be the requested value. In addition, if the corresponding saved limit can be
18562 represented correctly in an object of type **rlim_t** then it shall be overwritten with the new limit.

18563 The result of setting a limit to RLIM_SAVED_MAX or RLIM_SAVED_CUR is unspecified unless
18564 a previous call to *getrlimit()* returned that value as the soft or hard limit for the corresponding
18565 resource limit.

18566 The determination of whether a limit can be correctly represented in an object of type **rlim_t** is
18567 implementation-defined. For example, some implementations permit a limit whose value is
18568 greater than RLIM_INFINITY and others do not.

18569 The *exec* family of functions shall cause resource limits to be saved.

18570 RETURN VALUE

18571 Upon successful completion, *getrlimit()* and *setrlimit()* shall return 0. Otherwise, these functions
18572 shall return -1 and set *errno* to indicate the error.

18573 ERRORS

18574 The *getrlimit()* and *setrlimit()* functions shall fail if:

18575 [EINVAL] An invalid resource was specified; or in a *setrlimit()* call, the new *rlim_cur*
18576 exceeds the new *rlim_max*.

18577 [EPERM] The limit specified to *setrlimit()* would have raised the maximum limit value,
18578 and the calling process does not have appropriate privileges.

18579 The *setrlimit()* function may fail if:

18580 [EINVAL] The limit specified cannot be lowered because current usage is already higher
18581 than the limit.

18582 EXAMPLES

18583 None.

18584 APPLICATION USAGE

18585 If a process attempts to set the hard limit or soft limit for RLIMIT_NOFILE to less than the value
18586 of {_POSIX_OPEN_MAX} from <limits.h>, unexpected behavior may occur.

18587 If a process attempts to set the hard limit or soft limit for RLIMIT_NOFILE to less than the 1
18588 highest currently open file descriptor +1, unexpected behavior may occur. 1

18589 RATIONALE

18590 It should be noted that RLIMIT_STACK applies “at least” to the stack of the initial thread in the 2
18591 process, and not to the sum of all the stacks in the process, as that would be very limiting unless 2
18592 the value is so big as to provide no value at all with a single thread. 2

18593 FUTURE DIRECTIONS

18594 None.

18595 SEE ALSO

18596 exec, fork(), malloc(), open(), sigaltstack(), sysconf(), ulimit(), the Base Definitions volume of
18597 IEEE Std 1003.1-2001, <stropts.h>, <sys/resource.h>

18598 CHANGE HISTORY

18599 First released in Issue 4, Version 2.

18600 Issue 5

18601 Moved from X/OPEN UNIX extension to BASE.

18602 An APPLICATION USAGE section is added.

18603 Large File Summit extensions are added.

18604 Issue 6

18605 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/25 is applied, changing wording for 1
18606 RLIMIT_NOFILE in the DESCRIPTION related to functions that allocate a file descriptor failing 1
18607 with [EMFILE]. Text is added to the APPLICATION USAGE section noting the consequences of 1
18608 a process attempting to set the hard or soft limit for RLIMIT_NOFILE less than the highest 1
18609 currently open file descriptor +1. 1

18610 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/46 is applied, updating the definition of 2
18611 RLIMIT_STACK in the DESCRIPTION section from “the maximum size of a process stack” to 2
18612 “the maximum size of the initial thread’s stack”. Text is added to the RATIONALE section. 2

18613 **NAME**

18614 *getrusage* — get information about resource utilization

18615 **SYNOPSIS**

18616 XSI

```
#include <sys/resource.h>
```


18617

```
int getrusage(int who, struct rusage *r_usage);
```


18618

18619 **DESCRIPTION**

18620 The *getrusage*() function shall provide measures of the resources used by the current process or
18621 its terminated and waited-for child processes. If the value of the *who* argument is
18622 RUSAGE_SELF, information shall be returned about resources used by the current process. If the
18623 value of the *who* argument is RUSAGE_CHILDREN, information shall be returned about
18624 resources used by the terminated and waited-for children of the current process. If the child is
18625 never waited for (for example, if the parent has SA_NOCLDWAIT set or sets SIGCHLD to
18626 SIG_IGN), the resource information for the child process is discarded and not included in the
18627 resource information provided by *getrusage*().

18628 The *r_usage* argument is a pointer to an object of type **struct rusage** in which the returned
18629 information is stored.

18630 **RETURN VALUE**

18631 Upon successful completion, *getrusage*() shall return 0; otherwise, -1 shall be returned and *errno*
18632 set to indicate the error.

18633 **ERRORS**

18634 The *getrusage*() function shall fail if:

18635 [EINVAL] The value of the *who* argument is not valid.

18636 **EXAMPLES**18637 **Using getrusage()**

18638 The following example returns information about the resources used by the current process.

```
18639        #include <sys/resource.h>
18640        ...
18641        int who = RUSAGE_SELF;
18642        struct rusage usage;
18643        int ret;
18644        ret = getrusage(who, &usage);
```

18645 **APPLICATION USAGE**

18646 None.

18647 **RATIONALE**

18648 None.

18649 **FUTURE DIRECTIONS**

18650 None.

18651 **SEE ALSO**

18652 *exit()*, *sigaction()*, *time()*, *times()*, *wait()*, the Base Definitions volume of IEEE Std 1003.1-2001,
18653 <sys/resource.h>

18654 CHANGE HISTORY

18655 First released in Issue 4, Version 2.

18656 Issue 5

18657 Moved from X/OPEN UNIX extension to BASE.

18658 NAME

18659 gets — get a string from a stdin stream

18660 SYNOPSIS

```
18661        #include <stdio.h>
18662        char *gets(char *s);
```

18663 DESCRIPTION

18664 CX The functionality described on this reference page is aligned with the ISO C standard. Any
18665 conflict between the requirements described here and the ISO C standard is unintentional. This
18666 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

18667 The *gets()* function shall read bytes from the standard input stream, *stdin*, into the array pointed
18668 to by *s*, until a <newline> is read or an end-of-file condition is encountered. Any <newline> shall
18669 be discarded and a null byte shall be placed immediately after the last byte read into the array.

18670 CX The *gets()* function may mark the *st_atime* field of the file associated with *stream* for update. The
18671 *st_atime* field shall be marked for update by the first successful execution of *fgetc()*, *fgets()*,
18672 *fread()*, *getc()*, *getchar()*, *gets()*, *fscanf()*, or *scanf()* using *stream* that returns data not supplied by
18673 a prior call to *ungetc()*.

18674 RETURN VALUE

18675 Upon successful completion, *gets()* shall return *s*. If the stream is at end-of-file, the end-of-file
18676 indicator for the stream shall be set and *gets()* shall return a null pointer. If a read error occurs,
18677 CX the error indicator for the stream shall be set, *gets()* shall return a null pointer, and set *errno* to
18678 indicate the error.

18679 ERRORS

18680 Refer to *fgetc()*.

18681 EXAMPLES

18682 None.

18683 APPLICATION USAGE

18684 Reading a line that overflows the array pointed to by *s* results in undefined behavior. The use of
18685 *fgets()* is recommended.

18686 Since the user cannot specify the length of the buffer passed to *gets()*, use of this function is
18687 discouraged. The length of the string read is unlimited. It is possible to overflow this buffer in
18688 such a way as to cause applications to fail, or possible system security violations.

18689 It is recommended that the *fgets()* function should be used to read input lines.

18690 RATIONALE

18691 None.

18692 FUTURE DIRECTIONS

18693 None.

18694 SEE ALSO

18695 *feof()*, *ferror()*, *fgets()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**stdio.h**>

18696 CHANGE HISTORY

18697 First released in Issue 1. Derived from Issue 1 of the SVID.

18698 Issue 6

18699 Extensions beyond the ISO C standard are marked.

18700 NAME

18701 `getservbyname`, `getservbyport`, `getservent` — network services database functions

18702 SYNOPSIS

18703 `#include <netdb.h>`

18704 `struct servent *getservbyname(const char *name, const char *proto);`

18705 `struct servent *getservbyport(int port, const char *proto);`

18706 `struct servent *getservent(void);`

18707 DESCRIPTION

18708 Refer to `endservent()`.

18709 NAME

18710 *getsid* — get the process group ID of a session leader

18711 SYNOPSIS

18712 XSI `#include <unistd.h>`

18713 `pid_t getsid(pid_t pid);`

18714

18715 DESCRIPTION

18716 The *getsid()* function shall obtain the process group ID of the process that is the session leader of
18717 the process specified by *pid*. If *pid* is (**pid_t**)0, it specifies the calling process.

18718 RETURN VALUE

18719 Upon successful completion, *getsid()* shall return the process group ID of the session leader of
18720 the specified process. Otherwise, it shall return (**pid_t**)-1 and set *errno* to indicate the error.

18721 ERRORS

18722 The *getsid()* function shall fail if:

18723 [EPERM] The process specified by *pid* is not in the same session as the calling process,
18724 and the implementation does not allow access to the process group ID of the
18725 session leader of that process from the calling process.

18726 [ESRCH] There is no process with a process ID equal to *pid*.

18727 EXAMPLES

18728 None.

18729 APPLICATION USAGE

18730 None.

18731 RATIONALE

18732 None.

18733 FUTURE DIRECTIONS

18734 None.

18735 SEE ALSO

18736 *exec*, *fork()*, *getpid()*, *getpgid()*, *setpgid()*, *setsid()*, the Base Definitions volume of
18737 IEEE Std 1003.1-2001, `<unistd.h>`

18738 CHANGE HISTORY

18739 First released in Issue 4, Version 2.

18740 Issue 5

18741 Moved from X/OPEN UNIX extension to BASE.

18742 NAME

18743 *getsockname* — get the socket name

18744 SYNOPSIS

```
18745        #include <sys/socket.h>
18746        int getsockname(int socket, struct sockaddr *restrict address,
18747              socklen_t *restrict address_len);
```

18748 DESCRIPTION

18749 The *getsockname*() function shall retrieve the locally-bound name of the specified socket, store
18750 this address in the **sockaddr** structure pointed to by the *address* argument, and store the length of
18751 this address in the object pointed to by the *address_len* argument.

18752 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
18753 the stored address shall be truncated.

18754 If the socket has not been bound to a local name, the value stored in the object pointed to by
18755 *address* is unspecified.

18756 RETURN VALUE

18757 Upon successful completion, 0 shall be returned, the *address* argument shall point to the address
18758 of the socket, and the *address_len* argument shall point to the length of the address. Otherwise, -1
18759 shall be returned and *errno* set to indicate the error.

18760 ERRORS

18761 The *getsockname*() function shall fail if:

18762 [EBADF] The *socket* argument is not a valid file descriptor.

18763 [ENOTSOCK] The *socket* argument does not refer to a socket.

18764 [EOPNOTSUPP] The operation is not supported for this socket's protocol.

18765 The *getsockname*() function may fail if:

18766 [EINVAL] The socket has been shut down.

18767 [ENOBUFS] Insufficient resources were available in the system to complete the function.

18768 EXAMPLES

18769 None.

18770 APPLICATION USAGE

18771 None.

18772 RATIONALE

18773 None.

18774 FUTURE DIRECTIONS

18775 None.

18776 SEE ALSO

18777 *accept()*, *bind()*, *getpeername()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001,
18778 <sys/socket.h>

18779 CHANGE HISTORY

18780 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

18781 The **restrict** keyword is added to the *getsockname*() prototype for alignment with the
18782 ISO/IEC 9899: 1999 standard.

18783 NAME

18784 getsockopt — get the socket options

18785 SYNOPSIS

```
18786     #include <sys/socket.h>
18787
18788     int getsockopt(int socket, int level, int option_name,
18789           void *restrict option_value, socklen_t *restrict option_len);
```

18789 DESCRIPTION

18790 The *getsockopt()* function manipulates options associated with a socket.

18791 The *getsockopt()* function shall retrieve the value for the option specified by the *option_name* argument for the socket specified by the *socket* argument. If the size of the option value is greater than *option_len*, the value stored in the object pointed to by the *option_value* argument shall be silently truncated. Otherwise, the object pointed to by the *option_len* argument shall be modified to indicate the actual length of the value.

18796 The *level* argument specifies the protocol level at which the option resides. To retrieve options at the socket level, specify the *level* argument as SOL_SOCKET. To retrieve options at other levels, supply the appropriate level identifier for the protocol controlling the option. For example, to indicate that an option is interpreted by the TCP (Transmission Control Protocol), set *level* to IPPROTO_TCP as defined in the <netinet/in.h> header.

18801 The socket in use may require the process to have appropriate privileges to use the *getsockopt()* function.

18803 The *option_name* argument specifies a single option to be retrieved. It can be one of the following values defined in <sys/socket.h>:

18805 SO_DEBUG	Reports whether debugging information is being recorded. This option shall store an int value. This is a Boolean option.	
18807 SO_ACCEPTCONN	Reports whether socket listening is enabled. This option shall store an int value. This is a Boolean option.	
18809 SO_BROADCAST	Reports whether transmission of broadcast messages is supported, if this is supported by the protocol. This option shall store an int value. This is a Boolean option.	
18812 SO_REUSEADDR	Reports whether the rules used in validating addresses supplied to <i>bind()</i> should allow reuse of local addresses, if this is supported by the protocol. This option shall store an int value. This is a Boolean option.	
18815 SO_KEEPALIVE	Reports whether connections are kept active with periodic transmission of messages, if this is supported by the protocol.	
18817	If the connected socket fails to respond to these messages, the connection shall be broken and threads writing to that socket shall be notified with a SIGPIPE signal. This option shall store an int value. This is a Boolean option.	
18821 SO_LINGER	Reports whether the socket lingers on <i>close()</i> if data is present. If SO_LINGER is set, the system shall block the calling thread during <i>close()</i> until it can transmit the data or until the end of the interval indicated by the <i>l_linger</i> member, whichever comes first. If SO_LINGER is not specified, and <i>close()</i> is issued, the system handles the call in a way that allows the calling thread to continue as quickly as possible. This option shall store a linger structure.	2
18822		2
18823		2
18824		2
18825		2
18826		2
18827		2

18828	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.
18829		
18830		
18831	SO_SNDBUF	Reports send buffer size information. This option shall store an int value.
18832	SO_RCVBUF	Reports receive buffer size information. This option shall store an int value.
18833		
18834	SO_ERROR	Reports information about error status and clears it. This option shall store an int value.
18835		
18836	SO_TYPE	Reports the socket type. This option shall store an int value. Socket types are described in Section 2.10.6 (on page 60).
18837		
18838	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.
18839		
18840		
18841		
18842		
18843		
18844	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.
18845		
18846		
18847		
18848		
18849		
18850		
18851		
18852		
18853	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.
18854		
18855		
18856		
18857		
18858		
18859		
18860		
18861	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.
18862		
18863		
18864		
18865		
18866	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.
18867		
18868		
18869		
18870		
18871		
18872		
18873	For Boolean options, a zero value indicates that the option is disabled and a non-zero value indicates that the option is enabled.	
18874		

18875 RETURN VALUE

18876 Upon successful completion, *getsockopt()* shall return 0; otherwise, -1 shall be returned and *errno*
18877 set to indicate the error.

18878 ERRORS

18879 The *getsockopt()* function shall fail if:

18880 [EBADF] The *socket* argument is not a valid file descriptor.

18881 [EINVAL] The specified option is invalid at the specified socket level.

18882 [ENOPROTOOPT]

18883 The option is not supported by the protocol.

18884 [ENOTSOCK] The *socket* argument does not refer to a socket.

18885 The *getsockopt()* function may fail if:

18886 [EACCES] The calling process does not have the appropriate privileges.

18887 [EINVAL] The socket has been shut down.

18888 [ENOBUFS] Insufficient resources are available in the system to complete the function.

18889 EXAMPLES

18890 None.

18891 APPLICATION USAGE

18892 None.

18893 RATIONALE

18894 None.

18895 FUTURE DIRECTIONS

18896 None.

18897 SEE ALSO

18898 *bind()*, *close()*, *endprotoent()*, *setsockopt()*, *socket()*, the Base Definitions volume of
18899 IEEE Std 1003.1-2001, <sys/socket.h>, <netinet/in.h>

18900 CHANGE HISTORY

18901 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

18902 The **restrict** keyword is added to the *getsockopt()* prototype for alignment with the
18903 ISO/IEC 9899:1999 standard.

18904 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/47 is applied, updating the description of 2
18905 SO_LINGER in the DESCRIPTION so that it blocks the calling thread rather than the process. 2

18906 NAME

18907 getsubopt — parse suboption arguments from a string

18908 SYNOPSIS

18909 XSI #include <stdlib.h>

```
18910        int getsubopt(char **optionp, char * const *keylistp, char **valuep);
```

18911

18912 DESCRIPTION

18913 The *getsubopt()* function shall parse suboption arguments in a flag argument. Such options often
18914 result from the use of *getopt()*.

18915 The *getsubopt()* argument *optionp* is a pointer to a pointer to the option argument string. The suboption
18916 arguments shall be separated by commas and each may consist of either a single token,
18917 or a token-value pair separated by an equal sign.

18918 The *keylistp* argument shall be a pointer to a vector of strings. The end of the vector is identified
18919 by a null pointer. Each entry in the vector is one of the possible tokens that might be found in
18920 **optionp*. Since commas delimit suboption arguments in *optionp*, they should not appear in any of
18921 the strings pointed to by *keylistp*. Similarly, because an equal sign separates a token from its
18922 value, the application should not include an equal sign in any of the strings pointed to by
18923 *keylistp*.

18924 The *valuep* argument is the address of a value string pointer.

18925 If a comma appears in *optionp*, it shall be interpreted as a suboption separator. After commas
18926 have been processed, if there are one or more equal signs in a suboption string, the first equal
18927 sign in any suboption string shall be interpreted as a separator between a token and a value.
18928 Subsequent equal signs in a suboption string shall be interpreted as part of the value.

18929 If the string at **optionp* contains only one suboption argument (equivalently, no commas),
18930 *getsubopt()* shall update **optionp* to point to the null character at the end of the string. Otherwise,
18931 it shall isolate the suboption argument by replacing the comma separator with a null character,
18932 and shall update **optionp* to point to the start of the next suboption argument. If the suboption
18933 argument has an associated value (equivalently, contains an equal sign), *getsubopt()* shall update
18934 **valuep* to point to the value's first character. Otherwise, it shall set **valuep* to a null pointer. The
18935 calling application may use this information to determine whether the presence or absence of a
18936 value for the suboption is an error.

18937 Additionally, when *getsubopt()* fails to match the suboption argument with a token in the *keylistp*
18938 array, the calling application should decide if this is an error, or if the unrecognized option
18939 should be processed in another way.

18940 RETURN VALUE

18941 The *getsubopt()* function shall return the index of the matched token string, or -1 if no token
18942 strings were matched.

18943 ERRORS

18944 No errors are defined.

18945 EXAMPLES

```
18946     #include <stdio.h>
18947     #include <stdlib.h>
18948
18949     int do_all;
18950     const char *type;
18951     int read_size;
18952     int write_size;
18953     int read_only;
18954
18955     enum
18956     {
18955         RO_OPTION = 0,
18956         RW_OPTION,
18957         READ_SIZE_OPTION,
18958         WRITE_SIZE_OPTION
18959     };
18960
18961     const char *mount_opts[] =
18962     {
18963         [RO_OPTION] = "ro",
18964         [RW_OPTION] = "rw",
18965         [READ_SIZE_OPTION] = "rsize",
18966         [WRITE_SIZE_OPTION] = "wsize",
18967         NULL
18968     };
18969
18970     int
18971     main(int argc, char *argv[])
18972     {
18973         char *subopts, *value;
18974         int opt;
18975
18976         while ((opt = getopt(argc, argv, "at:o:")) != -1)
18977             switch(opt)
18978             {
18979                 case 'a':
18980                     do_all = 1;
18981                     break;
18982                 case 't':
18983                     type = optarg;
18984                     break;
18985                 case 'o':
18986                     subopts = optarg;
18987                     while (*subopts != '\0')
18988                         switch(getsubopt(&subopts, mount_opts, &value))
18989                         {
18990                             case RO_OPTION:
18991                                 read_only = 1;
18992                                 break;
18993                             case RW_OPTION:
18994                                 read_only = 0;
18995                                 break;
18996                             case READ_SIZE_OPTION:
```

```

18994         if (value == NULL)
18995             abort();
18996             read_size = atoi(value);
18997             break;
18998         case WRITE_SIZE_OPTION:
18999             if (value == NULL)
19000                 abort();
19001                 write_size = atoi(value);
19002                 break;
19003             default:
19004                 /* Unknown suboption. */
19005                 printf("Unknown suboption '%s'\n", value);
19006                 break;
19007             }
19008             break;
19009         default:
19010             abort();
19011         }
19012     /* Do the real work. */
19013     return 0;
19014 }
```

Parsing Suboptions

The following example uses the *getsubopt()* function to parse a *value* argument in the *optarg* external variable returned by a call to *getopt()*.

```

19018 #include <stdlib.h>
19019 ...
19020 char *tokens[] = {"HOME", "PATH", "LOGNAME", (char *) NULL };
19021 char *value;
19022 int opt, index;
19023 while ((opt = getopt(argc, argv, "e:")) != -1) {
19024     switch(opt) {
19025         case 'e' :
19026             while ((index = getsubopt(&optarg, tokens, &value)) != -1) {
19027                 switch(index) {
19028                     ...
19029                 }
19030                 break;
19031             ...
19032         }
19033     }
19034 }
```

APPLICATION USAGE

None.

RATIONALE

None.

19039 **FUTURE DIRECTIONS**

19040 None.

19041 **SEE ALSO**19042 *getopt()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>19043 **CHANGE HISTORY**

19044 First released in Issue 4, Version 2.

19045 **Issue 5**

19046 Moved from X/OPEN UNIX extension to BASE.

19047 **Issue 6**19048 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/26 is applied, correcting an editorial error 1
19049 in the SYNOPSIS. 1

19050 NAME

19051 `gettimeofday` — get the date and time

19052 SYNOPSIS

19053 XSI `#include <sys/time.h>`

19054 `int gettimeofday(struct timeval *restrict tp, void *restrict tzp);`

19055

19056 DESCRIPTION

19057 The `gettimeofday()` function shall obtain the current time, expressed as seconds and
19058 microseconds since the Epoch, and store it in the `timeval` structure pointed to by `tp`. The
19059 resolution of the system clock is unspecified.

19060 If `tzp` is not a null pointer, the behavior is unspecified.

19061 RETURN VALUE

19062 The `gettimeofday()` function shall return 0 and no value shall be reserved to indicate an error.

19063 ERRORS

19064 No errors are defined.

19065 EXAMPLES

19066 None.

19067 APPLICATION USAGE

19068 None.

19069 RATIONALE

19070 None.

19071 FUTURE DIRECTIONS

19072 None.

19073 SEE ALSO

19074 `ctime()`, `ftime()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/time.h>`

19075 CHANGE HISTORY

19076 First released in Issue 4, Version 2.

19077 Issue 5

19078 Moved from X/OPEN UNIX extension to BASE.

19079 Issue 6

19080 The DESCRIPTION is updated to refer to “seconds since the Epoch” rather than “seconds since
19081 00:00:00 UTC (Coordinated Universal Time), January 1 1970” for consistency with other `time`
19082 functions.

19083 The `restrict` keyword is added to the `gettimeofday()` prototype for alignment with the
19084 ISO/IEC 9899:1999 standard.

19085 NAME

19086 `getuid` — get a real user ID

19087 SYNOPSIS

```
19088        #include <unistd.h>
19089        uid_t getuid(void);
```

19090 DESCRIPTION

19091 The `getuid()` function shall return the real user ID of the calling process.

19092 RETURN VALUE

19093 The `getuid()` function shall always be successful and no return value is reserved to indicate the error.

19095 ERRORS

19096 No errors are defined.

19097 EXAMPLES**19098 Setting the Effective User ID to the Real User ID**

19099 The following example sets the effective user ID and the real user ID of the current process to the real user ID of the caller.

```
19101        #include <unistd.h>
19102        #include <sys/types.h>
19103        ...
19104        setreuid(getuid(), getuid());
19105        ...
```

19106 APPLICATION USAGE

19107 None.

19108 RATIONALE

19109 None.

19110 FUTURE DIRECTIONS

19111 None.

19112 SEE ALSO

19113 `getegid()`, `geteuid()`, `getgid()`, `setegid()`, `seteuid()`, `setgid()`, `setregid()`, `setreuid()`, `setuid()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/types.h>`, `<unistd.h>`

19115 CHANGE HISTORY

19116 First released in Issue 1. Derived from Issue 1 of the SVID.

19117 Issue 6

19118 In the SYNOPSIS, the optional include of the `<sys/types.h>` header is removed.

19119 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- 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.

19124 **NAME**

19125 `getutxent`, `getutxid`, `getutxline` — get user accounting database entries

19126 **SYNOPSIS**

19127 XSI `#include <utmpx.h>`

```
19128        struct utmpx *getutxent(void);
19129        struct utmpx *getutxid(const struct utmpx *id);
19130        struct utmpx *getutxline(const struct utmpx *line);
```

19131

19132 **DESCRIPTION**

19133 Refer to *endutxent()*.

19134 NAME

19135 getwc — get a wide character from a stream

19136 SYNOPSIS

```
19137 #include <stdio.h>
19138 #include <wchar.h>
19139 wint_t getwc(FILE *stream);
```

19140 DESCRIPTION

19141 CX The functionality described on this reference page is aligned with the ISO C standard. Any
19142 conflict between the requirements described here and the ISO C standard is unintentional. This
19143 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

19144 The *getwc()* function shall be equivalent to *fgetwc()*, except that if it is implemented as a macro it
19145 may evaluate *stream* more than once, so the argument should never be an expression with side
19146 effects.

19147 RETURN VALUE

19148 Refer to *fgetwc()*.

19149 ERRORS

19150 Refer to *fgetwc()*.

19151 EXAMPLES

19152 None.

19153 APPLICATION USAGE

19154 Since it may be implemented as a macro, *getwc()* may treat incorrectly a *stream* argument with
19155 side effects. In particular, *getwc(*f++)* does not necessarily work as expected. Therefore, use of
19156 this function is not recommended; *fgetwc()* should be used instead.

19157 RATIONALE

19158 None.

19159 FUTURE DIRECTIONS

19160 None.

19161 SEE ALSO

19162 *fgetwc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**stdio.h**>, <**wchar.h**>

19163 CHANGE HISTORY

19164 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
19165 draft.

19166 Issue 5

19167 The Optional Header (OH) marking is removed from <**stdio.h**>.

19168 NAME

19169 `getwchar` — get a wide character from a stdin stream

19170 SYNOPSIS

```
19171        #include <wchar.h>
19172        wint_t getwchar(void);
```

19173 DESCRIPTION

19174 CX The functionality described on this reference page is aligned with the ISO C standard. Any
19175 conflict between the requirements described here and the ISO C standard is unintentional. This
19176 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

19177 The `getwchar()` function shall be equivalent to `getwc(stdin)`.

19178 RETURN VALUE

19179 Refer to `fgetwc()`.

19180 ERRORS

19181 Refer to `fgetwc()`.

19182 EXAMPLES

19183 None.

19184 APPLICATION USAGE

19185 If the `wint_t` value returned by `getwchar()` is stored into a variable of type `wchar_t` and then
19186 compared against the `wint_t` macro `WEOF`, the result may be incorrect. Only the `wint_t` type is
19187 guaranteed to be able to represent any wide character and `WEOF`.

19188 RATIONALE

19189 None.

19190 FUTURE DIRECTIONS

19191 None.

19192 SEE ALSO

19193 `fgetwc()`, `getwc()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`

19194 CHANGE HISTORY

19195 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
19196 draft.

19197 NAME

19198 getwd — get the current working directory pathname (**LEGACY**)

19199 SYNOPSIS

19200 XSI #include <unistd.h>

19201 char *getwd(char *path_name);

19202

19203 DESCRIPTION

19204 The *getwd()* function shall determine an absolute pathname of the current working directory of
19205 the calling process, and copy a string containing that pathname into the array pointed to by the
19206 *path_name* argument.

19207 If the length of the pathname of the current working directory is greater than $\{\text{PATH_MAX}\}+1$
19208 including the null byte, *getwd()* shall fail and return a null pointer.

19209 RETURN VALUE

19210 Upon successful completion, a pointer to the string containing the absolute pathname of the
19211 current working directory shall be returned. Otherwise, *getwd()* shall return a null pointer and
19212 the contents of the array pointed to by *path_name* are undefined.

19213 ERRORS

19214 No errors are defined.

19215 EXAMPLES

19216 None.

19217 APPLICATION USAGE

19218 For applications portability, the *getcwd()* function should be used to determine the current
19219 working directory instead of *getwd()*.

19220 RATIONALE

19221 Since the user cannot specify the length of the buffer passed to *getwd()*, use of this function is
19222 discouraged. The length of a pathname described in $\{\text{PATH_MAX}\}$ is file system-dependent and
19223 may vary from one mount point to another, or might even be unlimited. It is possible to
19224 overflow this buffer in such a way as to cause applications to fail, or possible system security
19225 violations.

19226 It is recommended that the *getcwd()* function should be used to determine the current working
19227 directory.

19228 FUTURE DIRECTIONS

19229 This function may be withdrawn in a future version.

19230 SEE ALSO

19231 *getcwd()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

19232 CHANGE HISTORY

19233 First released in Issue 4, Version 2.

19234 Issue 5

19235 Moved from X/OPEN UNIX extension to BASE.

19236 Issue 6

19237 This function is marked LEGACY.

19238 NAME

19239 glob, globfree — generate pathnames matching a pattern

19240 SYNOPSIS

```
19241     #include <glob.h>
19242
19243     int glob(const char *restrict pattern, int flags,
19244             int(*errfunc)(const char *epath, int eerrno),
19245             glob_t *restrict pglob);
19246     void globfree(glob_t *pglob);
```

19246 DESCRIPTION

19247 The *glob()* function is a pathname generator that shall implement the rules defined in the Shell
 19248 and Utilities volume of IEEE Std 1003.1-2001, Section 2.13, Pattern Matching Notation, with
 19249 optional support for rule 3 in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section
 19250 2.13.3, Patterns Used for Filename Expansion.

19251 The structure type **glob_t** is defined in <**glob.h**> and includes at least the following members:

Member Type	Member Name	Description
size_t	<i>gl_pathc</i>	Count of paths matched by <i>pattern</i> .
char **	<i>gl_pathv</i>	Pointer to a list of matched filenames.
size_t	<i>gl_offs</i>	Slots to reserve at the beginning of <i>gl_pathv</i> .

19257 The argument *pattern* is a pointer to a pathname pattern to be expanded. The *glob()* function
 19258 shall match all accessible pathnames against this pattern and develop a list of all pathnames that
 19259 match. In order to have access to a pathname, *glob()* requires search permission on every
 19260 component of a path except the last, and read permission on each directory of any filename
 19261 component of *pattern* that contains any of the following special characters: '*', '?', and '['.

19262 The *glob()* function shall store the number of matched pathnames into *pglob->gl_pathc* and a
 19263 pointer to a list of pointers to pathnames into *pglob->gl_pathv*. The pathnames shall be in sort
 19264 order as defined by the current setting of the *LC_COLLATE* category; see the Base Definitions
 19265 volume of IEEE Std 1003.1-2001, Section 7.3.2, *LC_COLLATE*. The first pointer after the last
 19266 pathname shall be a null pointer. If the pattern does not match any pathnames, the returned
 19267 number of matched paths is set to 0, and the contents of *pglob->gl_pathv* are implementation-
 19268 defined.

19269 It is the caller's responsibility to create the structure pointed to by *pglob*. The *glob()* function shall
 19270 allocate other space as needed, including the memory pointed to by *gl_pathv*. The *globfree()*
 19271 function shall free any space associated with *pglob* from a previous call to *glob()*.

19272 The *flags* argument is used to control the behavior of *glob()*. The value of *flags* is a bitwise-
 19273 inclusive OR of zero or more of the following constants, which are defined in <**glob.h**>:

- | | |
|--------------------------|---|
| 19274 GLOB_APPEND | Append pathnames generated to the ones from a previous call to <i>glob()</i> . |
| 19275 GLOB_DOOFFS | Make use of <i>pglob->gl_offs</i> . If this flag is set, <i>pglob->gl_offs</i> is used to
19276 specify how many null pointers to add to the beginning of
19277 <i>pglob->gl_pathv</i> . In other words, <i>pglob->gl_pathv</i> shall point to
19278 <i>pglob->gl_offs</i> null pointers, followed by <i>pglob->gl_pathc</i> pathname
19279 pointers, followed by a null pointer. |
| 19280 GLOB_ERR | Cause <i>glob()</i> to return when it encounters a directory that it cannot open
19281 or read. Ordinarily, <i>glob()</i> continues to find matches. |

19282	GLOB_MARK	Each pathname that is a directory that matches <i>pattern</i> shall have a slash appended.
19283		
19284	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.
19285		
19286		
19287		
19288	GLOB_NOESCAPE	Disable backslash escaping.
19289	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.
19290		
19291		
19292		

The GLOB_APPEND flag can be used to append a new set of pathnames to those found in a previous call to *glob()*. The following rules apply to applications when two or more calls to *glob()* are made with the same value of *pglob* and without intervening calls to *globfree()*:

1. The first such call shall not set GLOB_APPEND. All subsequent calls shall set it.
2. All the calls shall set GLOB_DOOFFS, or all shall not set it.
3. After the second call, *pglob->gl_pathv* points to a list containing the following:
 - a. Zero or more null pointers, as specified by GLOB_DOOFFS and *pglob->gl_offs*.
 - b. Pointers to the pathnames that were in the *pglob->gl_pathv* list before the call, in the same order as before.
 - c. Pointers to the new pathnames generated by the second call, in the specified order.
4. The count returned in *pglob->gl_pathc* shall be the total number of pathnames from the two calls.
5. The application can change any of the fields after a call to *glob()*. If it does, the application shall reset them to the original value before a subsequent call, using the same *pglob* value, to *globfree()* or *glob()* with the GLOB_APPEND flag.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is not a null pointer, *glob()* calls (**errfunc*()) with two arguments:

1. The *epath* argument is a pointer to the path that failed.
2. The *errno* argument is the value of *errno* from the failure, as set by *opendir()*, *readdir()*, or *stat()*. (Other values may be used to report other errors not explicitly documented for those functions.)

If (**errfunc*()) is called and returns non-zero, or if the GLOB_ERR flag is set in *flags*, *glob()* shall stop the scan and return GLOB_ABORTED after setting *gl_pathc* and *gl_pathv* in *pglob* to reflect the paths already scanned. If GLOB_ERR is not set and either *errfunc* is a null pointer or (**errfunc*()) returns 0, the error shall be ignored.

The *glob()* function shall not fail because of large files.

19319 RETURN VALUE

Upon successful completion, *glob()* shall return 0. The argument *pglob->gl_pathc* shall return the number of matched pathnames and the argument *pglob->gl_pathv* shall contain a pointer to a null-terminated list of matched and sorted pathnames. However, if *pglob->gl_pathc* is 0, the content of *pglob->gl_pathv* is undefined.

19324 The *globfree()* function shall not return a value.
 19325 If *glob()* terminates due to an error, it shall return one of the non-zero constants defined in
 19326 <glob.h>. The arguments *pglob->gl_pathc* and *pglob->gl_pathv* are still set as defined above.

19327 ERRORS

19328 The *glob()* function shall fail and return the corresponding value if:
 19329 GLOB_ABORTED The scan was stopped because GLOB_ERR was set or (**errfunc()*)
 19330 returned non-zero.
 19331 GLOB_NOMATCH The pattern does not match any existing pathname, and
 19332 GLOB_NOCHECK was not set in flags.
 19333 GLOB_NOSPACE An attempt to allocate memory failed.

19334 EXAMPLES

19335 One use of the GLOB_DOOFFS flag is by applications that build an argument list for use with
 19336 *execv()*, *execve()*, or *execvp()*. Suppose, for example, that an application wants to do the
 19337 equivalent of:

19338 ls -l *.c

19339 but for some reason:

19340 system("ls -l *.c")

19341 is not acceptable. The application could obtain approximately the same result using the
 19342 sequence:

```
19343        globbuf.gl_offs = 2;
19344        glob("*.c", GLOB_DOOFFS, NULL, &globbuf);
19345        globbuf.gl_pathv[0] = "ls";
19346        globbuf.gl_pathv[1] = "-l";
19347        execvp("ls", &globbuf.gl_pathv[0]);
```

19348 Using the same example:

19349 ls -l *.c *.h

19350 could be approximately simulated using GLOB_APPEND as follows:

```
19351        globbuf.gl_offs = 2;
19352        glob("*.c", GLOB_DOOFFS, NULL, &globbuf);
19353        glob("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);
19354        ...
```

19355 APPLICATION USAGE

19356 This function is not provided for the purpose of enabling utilities to perform pathname
 19357 expansion on their arguments, as this operation is performed by the shell, and utilities are
 19358 explicitly not expected to redo this. Instead, it is provided for applications that need to do
 19359 pathname expansion on strings obtained from other sources, such as a pattern typed by a user or
 19360 read from a file.

19361 If a utility needs to see if a pathname matches a given pattern, it can use *fnmatch()*.

19362 Note that *gl_pathc* and *gl_pathv* have meaning even if *glob()* fails. This allows *glob()* to report
 19363 partial results in the event of an error. However, if *gl_pathc* is 0, *gl_pathv* is unspecified even if
 19364 *glob()* did not return an error.

19365 The GLOB_NOCHECK option could be used when an application wants to expand a pathname
 19366 if wildcards are specified, but wants to treat the pattern as just a string otherwise. The *sh* utility

19367 might use this for option-arguments, for example.
19368 The new pathnames generated by a subsequent call with GLOB_APPEND are not sorted
19369 together with the previous pathnames. This mirrors the way that the shell handles pathname
19370 expansion when multiple expansions are done on a command line.
19371 Applications that need tilde and parameter expansion should use *wordexp()*.

19372 RATIONALE

19373 It was claimed that the GLOB_DOOFFS flag is unnecessary because it could be simulated using:

```
19374     new = (char **)malloc((n + pglob->gl_pathc + 1)
19375             * sizeof(char *));
19376     (void) memcpy(new+n, pglob->gl_pathv,
19377             pglob->gl_pathc * sizeof(char *));
19378     (void) memset(new, 0, n * sizeof(char *));
19379     free(pglob->gl_pathv);
19380     pglob->gl_pathv = new;
```

19381 However, this assumes that the memory pointed to by *gl_pathv* is a block that was separately
19382 created using *malloc()*. This is not necessarily the case. An application should make no
19383 assumptions about how the memory referenced by fields in *pglob* was allocated. It might have
19384 been obtained from *malloc()* in a large chunk and then carved up within *glob()*, or it might have
19385 been created using a different memory allocator. It is not the intent of the standard developers to
19386 specify or imply how the memory used by *glob()* is managed.

19387 The GLOB_APPEND flag would be used when an application wants to expand several different
19388 patterns into a single list.

19389 FUTURE DIRECTIONS

19390 None.

19391 SEE ALSO

19392 *exec*, *fnmatch()*, *opendir()*, *readdir()*, *stat()*, *wordexp()*, the Base Definitions volume of
19393 IEEE Std 1003.1-2001, <*glob.h*>, the Shell and Utilities volume of IEEE Std 1003.1-2001

19394 CHANGE HISTORY

19395 First released in Issue 4. Derived from the ISO POSIX-2 standard.

19396 Issue 5

19397 Moved from POSIX2 C-language Binding to BASE.

19398 Issue 6

19399 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

19400 The **restrict** keyword is added to the *glob()* prototype for alignment with the ISO/IEC 9899:1999
19401 standard.

19402 NAME

19403 `gmtime`, `gmtime_r` — convert a time value to a broken-down UTC time

19404 SYNOPSIS

```
19405       #include <time.h>
19406       struct tm *gmtime(const time_t *timer);
19407 TSF       struct tm *gmtime_r(const time_t *restrict timer,
19408                    struct tm *restrict result);
19409
```

19410 DESCRIPTION

19411 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.

19414 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).

19416 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.

19421 TSF The same relationship shall apply for `gmtime_r()`.

19422 CX The `gmtime()` function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

19424 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.

19428 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.

19432 RETURN VALUE

19433 Upon successful completion, the `gmtime()` function shall return a pointer to a `struct tm`. If an error is detected, `gmtime()` shall return a null pointer and set `errno` to indicate the error. 1

19435 TSF Upon successful completion, `gmtime_r()` shall return the address of the structure pointed to by the argument *result*. If an error is detected, `gmtime_r()` shall return a null pointer and set `errno` to indicate the error. 2

19438 ERRORS

19439 TSF The `gmtime()` and `gmtime_r()` functions shall fail if: 2

19440 CX [EOVERFLOW] The result cannot be represented. 2

19441 EXAMPLES

19442 None.

19443 APPLICATION USAGE

19444 The *gmtime_r()* function is thread-safe and returns values in a user-supplied buffer instead of
19445 possibly using a static data area that may be overwritten by each call.

19446 RATIONALE

19447 None.

19448 FUTURE DIRECTIONS

19449 None.

19450 SEE ALSO

19451 *asctime()*, *clock()*, *ctime()*, *difftime()*, *localtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*, *utime()*,
19452 the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

19453 CHANGE HISTORY

19454 First released in Issue 1. Derived from Issue 1 of the SVID.

19455 Issue 5

19456 A note indicating that the *gmtime()* function need not be reentrant is added to the
19457 DESCRIPTION.19458 The *gmtime_r()* function is included for alignment with the POSIX Threads Extension.

19459 Issue 6

19460 The *gmtime_r()* function is marked as part of the Thread-Safe Functions option.

19461 Extensions beyond the ISO C standard are marked.

19462 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
19463 its avoidance of possibly using a static data area.19464 The **restrict** keyword is added to the *gmtime_r()* prototype for alignment with the
19465 ISO/IEC 9899:1999 standard.19466 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/27 is applied, adding the [EOVERFLOW] 1
19467 error. 119468 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/48 is applied, updating the error handling 2
19469 for *gmtime_r()*. 2

19470 NAME

19471 grantpt — grant access to the slave pseudo-terminal device

19472 SYNOPSIS

19473 XSI #include <stdlib.h>

19474 int grantpt(int *fildes*);

19475

19476 DESCRIPTION

19477 The *grantpt()* function shall change the mode and ownership of the slave pseudo-terminal
19478 device associated with its master pseudo-terminal counterpart. The *fildes* argument is a file
19479 descriptor that refers to a master pseudo-terminal device. The user ID of the slave shall be set to
19480 the real UID of the calling process and the group ID shall be set to an unspecified group ID. The
19481 permission mode of the slave pseudo-terminal shall be set to readable and writable by the
19482 owner, and writable by the group.

19483 The behavior of the *grantpt()* function is unspecified if the application has installed a signal
19484 handler to catch SIGCHLD signals.

19485 RETURN VALUE

19486 Upon successful completion, *grantpt()* shall return 0; otherwise, it shall return -1 and set *errno* to
19487 indicate the error.

19488 ERRORS

19489 The *grantpt()* function may fail if:

19490 [EBADF] The *fildes* argument is not a valid open file descriptor.

19491 [EINVAL] The *fildes* argument is not associated with a master pseudo-terminal device.

19492 [EACCES] The corresponding slave pseudo-terminal device could not be accessed.

19493 EXAMPLES

19494 None.

19495 APPLICATION USAGE

19496 None.

19497 RATIONALE

19498 None.

19499 FUTURE DIRECTIONS

19500 None.

19501 SEE ALSO

19502 *open()*, *ptsname()*, *unlockpt()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

19503 CHANGE HISTORY

19504 First released in Issue 4, Version 2.

19505 Issue 5

19506 Moved from X/OPEN UNIX extension to BASE.

19507 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.

19508 NAME

19509 *h_errno* — error return value for network database operations

19510 SYNOPSIS

19511 OB `#include <netdb.h>`
19512

19513 DESCRIPTION

19514 This method of returning errors is used only in connection with obsolescent functions.

19515 The `<netdb.h>` header provides a declaration of *h_errno* as a modifiable lvalue of type **int**.

19516 It is unspecified whether *h_errno* is a macro or an identifier declared with external linkage. If a
19517 macro definition is suppressed in order to access an actual object, or a program defines an
19518 identifier with the name *h_errno*, the behavior is undefined.

19519 RETURN VALUE

19520 None.

19521 ERRORS

19522 No errors are defined.

19523 EXAMPLES

19524 None.

19525 APPLICATION USAGE

19526 Applications should obtain the definition of *h_errno* by the inclusion of the `<netdb.h>` header.

19527 RATIONALE

19528 None.

19529 FUTURE DIRECTIONS

19530 *h_errno* may be withdrawn in a future version.

19531 SEE ALSO

19532 `endhostent()`, *errno*, the Base Definitions volume of IEEE Std 1003.1-2001, `<netdb.h>`

19533 CHANGE HISTORY

19534 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19535 NAME

19536 hcreate, hdestroy, hsearch — manage hash search table

19537 SYNOPSIS

```
19538 XSI #include <search.h>
19539     int hcreate(size_t nel);
19540     void hdestroy(void);
19541     ENTRY *hsearch(ENTRY item, ACTION action);
19542
```

19543 DESCRIPTION

19544 The *hcreate()*, *hdestroy()*, and *hsearch()* functions shall manage hash search tables.

19545 The *hcreate()* function shall allocate sufficient space for the table, and the application shall
 19546 ensure it is called before *hsearch()* is used. The *nel* argument is an estimate of the maximum
 19547 number of entries that the table shall contain. This number may be adjusted upward by the
 19548 algorithm in order to obtain certain mathematically favorable circumstances.

19549 The *hdestroy()* function shall dispose of the search table, and may be followed by another call to
 19550 *hcreate()*. After the call to *hdestroy()*, the data can no longer be considered accessible.

19551 The *hsearch()* function is a hash-table search routine. It shall return a pointer into a hash table
 19552 indicating the location at which an entry can be found. The *item* argument is a structure of type
 19553 **ENTRY** (defined in the **<search.h>** header) containing two pointers: *item.key* points to the
 19554 comparison key (a **char ***), and *item.data* (a **void ***) points to any other data to be associated with
 19555 that key. The comparison function used by *hsearch()* is *strcmp()*. The *action* argument is a
 19556 member of an enumeration type **ACTION** indicating the disposition of the entry if it cannot be
 19557 found in the table. **ENTER** indicates that the item should be inserted in the table at an
 19558 appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is
 19559 indicated by the return of a null pointer.

19560 These functions need not be reentrant. A function that is not required to be reentrant is not
 19561 required to be thread-safe.

19562 RETURN VALUE

19563 The *hcreate()* function shall return 0 if it cannot allocate sufficient space for the table; otherwise,
 19564 it shall return non-zero.

19565 The *hdestroy()* function shall not return a value.

19566 The *hsearch()* function shall return a null pointer if either the action is **FIND** and the item could
 19567 not be found or the action is **ENTER** and the table is full.

19568 ERRORS

19569 The *hcreate()* and *hsearch()* functions may fail if:

19570 [ENOMEM] Insufficient storage space is available.

19571 EXAMPLES

19572 The following example reads in strings followed by two numbers and stores them in a hash
 19573 table, discarding duplicates. It then reads in strings and finds the matching entry in the hash
 19574 table and prints it out.

```
19575 #include <stdio.h>
19576 #include <search.h>
19577 #include <string.h>
19578
19579 struct info {           /* This is the info stored in the table */
19580     int age, room;      /* other than the key. */
```

```
19580     };
19581     #define NUM_EMPL      5000      /* # of elements in search table. */
19582     int main(void)
19583     {
19584         char string_space[NUM_EMPL*20];    /* Space to store strings. */
19585         struct info info_space[NUM_EMPL]; /* Space to store employee info. */
19586         char *str_ptr = string_space;      /* Next space in string_space. */
19587         struct info *info_ptr = info_space;
19588                                         /* Next space in info_space. */
19589         ENTRY item;
19590         ENTRY *found_item; /* Name to look for in table. */
19591         char name_to_find[30];
19592         int i = 0;
19593         /* Create table; no error checking is performed. */
19594         (void) hcreate(NUM_EMPL);
19595         while (scanf("%s%d%d", str_ptr, &info_ptr->age,
19596                         &info_ptr->room) != EOF && i++ < NUM_EMPL) {
19597             /* Put information in structure, and structure in item. */
19598             item.key = str_ptr;
19599             item.data = info_ptr;
19600             str_ptr += strlen(str_ptr) + 1;
19601             info_ptr++;
19602             /* Put item into table. */
19603             (void) hsearch(item, ENTER);
19604         }
19605         /* Access table. */
19606         item.key = name_to_find;
19607         while (scanf("%s", item.key) != EOF) {
19608             if ((found_item = hsearch(item, FIND)) != NULL) {
19609                 /* If item is in the table. */
19610                 (void)printf("found %s, age = %d, room = %d\n",
19611                             found_item->key,
19612                             ((struct info *)found_item->data)->age,
19613                             ((struct info *)found_item->data)->room);
19614             } else
19615                 (void)printf("no such employee %s\n", name_to_find);
19616         }
19617         return 0;
19618     }
```

19619 APPLICATION USAGE

19620 The *hcreate()* and *hsearch()* functions may use *malloc()* to allocate space.

19621 RATIONALE

19622 None.

19623 FUTURE DIRECTIONS

19624 None.

19625 SEE ALSO

19626 *bsearch()*, *lsearch()*, *malloc()*, *strcmp()*, *tsearch()*, the Base Definitions volume of
19627 IEEE Std 1003.1-2001, <search.h>

19628 CHANGE HISTORY

19629 First released in Issue 1. Derived from Issue 1 of the SVID.

19630 Issue 6

19631 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

19632 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

19633 NAME

19634 `htonl`, `htons`, `ntohl`, `ntohs` — convert values between host and network byte order

19635 SYNOPSIS

```
19636        #include <arpa/inet.h>
19637        uint32_t htonl(uint32_t hostlong);
19638        uint16_t htons(uint16_t hostshort);
19639        uint32_t ntohl(uint32_t netlong);
19640        uint16_t ntohs(uint16_t netshort);
```

19641 DESCRIPTION

19642 These functions shall convert 16-bit and 32-bit quantities between network byte order and host
19643 byte order.

19644 On some implementations, these functions are defined as macros.

19645 The `uint32_t` and `uint16_t` types are defined in `<inttypes.h>`.

19646 RETURN VALUE

19647 The `htonl()` and `htons()` functions shall return the argument value converted from host to
19648 network byte order.

19649 The `ntohl()` and `ntohs()` functions shall return the argument value converted from network to
19650 host byte order.

19651 ERRORS

19652 No errors are defined.

19653 EXAMPLES

19654 None.

19655 APPLICATION USAGE

19656 These functions are most often used in conjunction with IPv4 addresses and ports as returned by
19657 `gethostent()` and `getservent()`.

19658 RATIONALE

19659 None.

19660 FUTURE DIRECTIONS

19661 None.

19662 SEE ALSO

19663 `endhostent()`, `endservent()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<inttypes.h>`,
19664 `<arpa/inet.h>`

19665 CHANGE HISTORY

19666 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19667 NAME

19668 hypot, hypotf, hypotl — Euclidean distance function

19669 SYNOPSIS

```
19670     #include <math.h>
19671
19672     double hypot(double x, double y);
19673     float hypotf(float x, float y);
19674     long double hypotl(long double x, long double y);
```

19674 DESCRIPTION

19675 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 19676 conflict between the requirements described here and the ISO C standard is unintentional. This
 19677 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

19678 These functions shall compute the value of the square root of x^2+y^2 without undue overflow or
 19679 underflow.

19680 An application wishing to check for error situations should set *errno* to zero and call
 19681 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
 19682 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
 19683 zero, an error has occurred.

19684 RETURN VALUE

19685 Upon successful completion, these functions shall return the length of the hypotenuse of a
 19686 right-angled triangle with sides of length *x* and *y*.

19687 If the correct value would cause overflow, a range error shall occur and *hypot()*, *hypotf()*, and
 19688 *hypotl()* shall return the value of the macro *HUGE_VAL*, *HUGE_VALF*, and *HUGE_VALL*,
 19689 respectively.

19690 MX If *x* or *y* is $\pm\text{Inf}$, $+\text{Inf}$ shall be returned (even if one of *x* or *y* is *NaN*).

19691 If *x* or *y* is *NaN*, and the other is not $\pm\text{Inf}$, a *NaN* shall be returned.

19692 If both arguments are subnormal and the correct result is subnormal, a range error may occur
 19693 and the correct result is returned.

19694 ERRORS

19695 These functions shall fail if:

19696 Range Error The result overflows.

19697 If the integer expression (*math_errhandling* & *MATH_ERRNO*) is non-zero,
 19698 then *errno* shall be set to [ERANGE]. If the integer expression
 19699 (*math_errhandling* & *MATH_ERREXCEPT*) is non-zero, then the overflow
 19700 floating-point exception shall be raised.

19701 These functions may fail if:

19702 MX Range Error The result underflows.

19703 If the integer expression (*math_errhandling* & *MATH_ERRNO*) is non-zero,
 19704 then *errno* shall be set to [ERANGE]. If the integer expression
 19705 (*math_errhandling* & *MATH_ERREXCEPT*) is non-zero, then the underflow
 19706 floating-point exception shall be raised.

19707 EXAMPLES

19708 See the EXAMPLES section in *atan2()*.

2

19709 APPLICATION USAGE

19710 *hypot(x,y)*, *hypot(y,x)*, and *hypot(x,-y)* are equivalent.

19711 *hypot(x,±0)* is equivalent to *fabs(x)*.

19712 Underflow only happens when both *x* and *y* are subnormal and the (inexact) result is also subnormal.

19714 These functions take precautions against overflow during intermediate steps of the computation.

19716 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

19718 RATIONALE

19719 None.

19720 FUTURE DIRECTIONS

19721 None.

19722 SEE ALSO

19723 *atan2()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *sqrt()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, 2
19724
19725 *<math.h>*

19726 CHANGE HISTORY

19727 First released in Issue 1. Derived from Issue 1 of the SVID.

19728 Issue 5

19729 The DESCRIPTION is updated to indicate how an application should check for an error. This
19730 text was previously published in the APPLICATION USAGE section.

19731 Issue 6

19732 The *hypot()* function is no longer marked as an extension.

19733 The *hypotf()* and *hypotl()* functions are added for alignment with the ISO/IEC 9899:1999
19734 standard.

19735 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
19736 revised to align with the ISO/IEC 9899:1999 standard.

19737 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
19738 marked.

19739 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/49 is applied, updating the EXAMPLES 2
19740 section.

19741 NAME

19742 iconv — codeset conversion function

19743 SYNOPSIS

19744 XSI #include <iconv.h>

```
19745 size_t iconv(iconv_t cd, char **restrict inbuf,  
19746     size_t *restrict inbytesleft, char **restrict outbuf,  
19747     size_t *restrict outbytesleft);  
19748
```

19749 DESCRIPTION

19750 The *iconv()* function shall convert the sequence of characters from one codeset, in the array
19751 specified by *inbuf*, into a sequence of corresponding characters in another codeset, in the array
19752 specified by *outbuf*. The codesets are those specified in the *iconv_open()* call that returned the
19753 conversion descriptor, *cd*. The *inbuf* argument points to a variable that points to the first
19754 character in the input buffer and *inbytesleft* indicates the number of bytes to the end of the buffer
19755 to be converted. The *outbuf* argument points to a variable that points to the first available byte in
19756 the output buffer and *outbytesleft* indicates the number of the available bytes to the end of the
19757 buffer.

19758 For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by
19759 a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When *iconv()* is
19760 called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft*
19761 points to a positive value, *iconv()* shall place, into the output buffer, the byte sequence to change
19762 the output buffer to its initial shift state. If the output buffer is not large enough to hold the
19763 entire reset sequence, *iconv()* shall fail and set *errno* to [E2BIG]. Subsequent calls with *inbuf* as
19764 other than a null pointer or a pointer to a null pointer cause the conversion to take place from
19765 the current state of the conversion descriptor.

19766 If a sequence of input bytes does not form a valid character in the specified codeset, conversion
19767 shall stop after the previous successfully converted character. If the input buffer ends with an
19768 incomplete character or shift sequence, conversion shall stop after the previous successfully
19769 converted bytes. If the output buffer is not large enough to hold the entire converted input,
19770 conversion shall stop just prior to the input bytes that would cause the output buffer to
19771 overflow. The variable pointed to by *inbuf* shall be updated to point to the byte following the last
19772 byte successfully used in the conversion. The value pointed to by *inbytesleft* shall be
19773 decremented to reflect the number of bytes still not converted in the input buffer. The variable
19774 pointed to by *outbuf* shall be updated to point to the byte following the last byte of converted
19775 output data. The value pointed to by *outbytesleft* shall be decremented to reflect the number of
19776 bytes still available in the output buffer. For state-dependent encodings, the conversion
19777 descriptor shall be updated to reflect the shift state in effect at the end of the last successfully
19778 converted byte sequence.

19779 If *iconv()* encounters a character in the input buffer that is valid, but for which an identical
19780 character does not exist in the target codeset, *iconv()* shall perform an implementation-defined
19781 conversion on this character.

19782 RETURN VALUE

19783 The *iconv()* function shall update the variables pointed to by the arguments to reflect the extent
19784 of the conversion and return the number of non-identical conversions performed. If the entire
19785 string in the input buffer is converted, the value pointed to by *inbytesleft* shall be 0. If the input
19786 conversion is stopped due to any conditions mentioned above, the value pointed to by *inbytesleft*
19787 shall be non-zero and *errno* shall be set to indicate the condition. If an error occurs, *iconv()* shall
19788 return (*size_t*)–1 and set *errno* to indicate the error.

19789 **ERRORS**

19790 The *iconv()* function shall fail if:

19791 [EILSEQ] Input conversion stopped due to an input byte that does not belong to the
19792 input codeset.

19793 [E2BIG] Input conversion stopped due to lack of space in the output buffer.

19794 [EINVAL] Input conversion stopped due to an incomplete character or shift sequence at
19795 the end of the input buffer.

19796 The *iconv()* function may fail if:

19797 [EBADF] The *cd* argument is not a valid open conversion descriptor.

19798 **EXAMPLES**

19799 None.

19800 **APPLICATION USAGE**

19801 The *inbuf* argument indirectly points to the memory area which contains the conversion input
19802 data. The *outbuf* argument indirectly points to the memory area which is to contain the result of
19803 the conversion. The objects indirectly pointed to by *inbuf* and *outbuf* are not restricted to
19804 containing data that is directly representable in the ISO C standard language **char** data type. The
19805 type of *inbuf* and *outbuf*, **char** **, does not imply that the objects pointed to are interpreted as
19806 null-terminated C strings or arrays of characters. Any interpretation of a byte sequence that
19807 represents a character in a given character set encoding scheme is done internally within the
19808 codeset converters. For example, the area pointed to indirectly by *inbuf* and/or *outbuf* can
19809 contain all zero octets that are not interpreted as string terminators but as coded character data
19810 according to the respective codeset encoding scheme. The type of the data (**char**, **short**, **long**, and
19811 so on) read or stored in the objects is not specified, but may be inferred for both the input and
19812 output data by the converters determined by the *fromcode* and *tocode* arguments of *iconv_open()*.

19813 Regardless of the data type inferred by the converter, the size of the remaining space in both
19814 input and output objects (the *inbytesleft* and *outbytesleft* arguments) is always measured in bytes.

19815 For implementations that support the conversion of state-dependent encodings, the conversion
19816 descriptor must be able to accurately reflect the shift-state in effect at the end of the last
19817 successful conversion. It is not required that the conversion descriptor itself be updated, which
19818 would require it to be a pointer type. Thus, implementations are free to implement the
19819 descriptor as a handle (other than a pointer type) by which the conversion information can be
19820 accessed and updated.

19821 **RATIONALE**

19822 None.

19823 **FUTURE DIRECTIONS**

19824 None.

19825 **SEE ALSO**

19826 *iconv_open()*, *iconv_close()*, the Base Definitions volume of IEEE Std 1003.1-2001, <iconv.h>

19827 **CHANGE HISTORY**

19828 First released in Issue 4. Derived from the HP-UX Manual.

19829 **Issue 6**

19830 The SYNOPSIS has been corrected to align with the <iconv.h> reference page.

19831 The **restrict** keyword is added to the *iconv()* prototype for alignment with the
19832 ISO/IEC 9899:1999 standard.

19833 NAME

19834 *iconv_close* — codeset conversion deallocation function

19835 SYNOPSIS

19836 XSI `#include <iconv.h>`

19837 `int iconv_close(iconv_t cd);`

19838

19839 DESCRIPTION

19840 The *iconv_close()* function shall deallocate the conversion descriptor *cd* and all other associated
19841 resources allocated by *iconv_open()*.

19842 If a file descriptor is used to implement the type **iconv_t**, that file descriptor shall be closed.

19843 RETURN VALUE

19844 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
19845 indicate the error.

19846 ERRORS

19847 The *iconv_close()* function may fail if:

19848 [EBADF] The conversion descriptor is invalid.

19849 EXAMPLES

19850 None.

19851 APPLICATION USAGE

19852 None.

19853 RATIONALE

19854 None.

19855 FUTURE DIRECTIONS

19856 None.

19857 SEE ALSO

19858 *iconv()*, *iconv_open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**iconv.h**>

19859 CHANGE HISTORY

19860 First released in Issue 4. Derived from the HP-UX Manual.

19861 NAME

19862 **iconv_open** — codeset conversion allocation function

19863 SYNOPSIS

19864 XSI **#include <iconv.h>**

19865 **iconv_t iconv_open(const char *tocode, const char *fromcode);**

19866

19867 DESCRIPTION

19868 The *iconv_open()* function shall return a conversion descriptor that describes a conversion from
19869 the codeset specified by the string pointed to by the *fromcode* argument to the codeset specified
19870 by the string pointed to by the *tocode* argument. For state-dependent encodings, the conversion
19871 descriptor shall be in a codeset-dependent initial shift state, ready for immediate use with
19872 *iconv()*.

19873 Settings of *fromcode* and *tocode* and their permitted combinations are implementation-defined.

19874 A conversion descriptor shall remain valid until it is closed by *iconv_close()* or an implicit close.

19875 If a file descriptor is used to implement conversion descriptors, the FD_CLOEXEC flag shall be
19876 set; see <**fcntl.h**>.

19877 RETURN VALUE

19878 Upon successful completion, *iconv_open()* shall return a conversion descriptor for use on
19879 subsequent calls to *iconv()*. Otherwise, *iconv_open()* shall return (**iconv_t**)−1 and set *errno* to
19880 indicate the error.

19881 ERRORS

19882 The *iconv_open()* function may fail if:

19883 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

19884 [ENFILE] Too many files are currently open in the system.

19885 [ENOMEM] Insufficient storage space is available.

19886 [EINVAL] The conversion specified by *fromcode* and *tocode* is not supported by the
19887 implementation.

19888 EXAMPLES

19889 None.

19890 APPLICATION USAGE

19891 Some implementations of *iconv_open()* use *malloc()* to allocate space for internal buffer areas.
19892 The *iconv_open()* function may fail if there is insufficient storage space to accommodate these
19893 buffers.

19894 Conforming applications must assume that conversion descriptors are not valid after a call to
19895 one of the *exec* functions.

19896 Application developers should consult the system documentation to determine the supported
19897 codesets and their naming schemes.

19898 RATIONALE

19899 None.

19900 FUTURE DIRECTIONS

19901 None.

19902 **SEE ALSO**

19903 *iconv()*, *iconv_close()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>, <iconv.h>

19904 **CHANGE HISTORY**

19905 First released in Issue 4. Derived from the HP-UX Manual.

19906 NAME

19907 *if_freenameindex* — free memory allocated by *if_nameindex*

19908 SYNOPSIS

```
19909        #include <net/if.h>
19910        void if_freenameindex(struct if_nameindex *ptr);
```

19911 DESCRIPTION

19912 The *if_freenameindex()* function shall free the memory allocated by *if_nameindex()*. The *ptr* argument shall be a pointer that was returned by *if_nameindex()*. After *if_freenameindex()* has been called, the application shall not use the array of which *ptr* is the address.

19915 RETURN VALUE

19916 None.

19917 ERRORS

19918 No errors are defined.

19919 EXAMPLES

19920 None.

19921 APPLICATION USAGE

19922 None.

19923 RATIONALE

19924 None.

19925 FUTURE DIRECTIONS

19926 None.

19927 SEE ALSO

19928 *getsockopt()*, *if_indextoname()*, *if_nameindex()*, *if_nametoindex()*, *setsockopt()*, the Base Definitions
19929 volume of IEEE Std 1003.1-2001, <net/if.h>

19930 CHANGE HISTORY

19931 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19932 NAME

19933 *if_indextoname* — map a network interface index to its corresponding name

19934 SYNOPSIS

```
19935        #include <net/if.h>
19936        char *if_indextoname(unsigned ifindex, char *ifname);
```

19937 DESCRIPTION

19938 The *if_indextoname()* function shall map an interface index to its corresponding name.

19939 When this function is called, *ifname* shall point to a buffer of at least {IF_NAMESIZE} bytes. The 1
19940 function shall place in this buffer the name of the interface with index *ifindex*.

19941 RETURN VALUE

19942 If *ifindex* is an interface index, then the function shall return the value supplied in *ifname*, which
19943 points to a buffer now containing the interface name. Otherwise, the function shall return a
19944 NULL pointer and set *errno* to indicate the error.

19945 ERRORS

19946 The *if_indextoname()* function shall fail if:

19947 [ENXIO] The interface does not exist.

19948 EXAMPLES

19949 None.

19950 APPLICATION USAGE

19951 None.

19952 RATIONALE

19953 None.

19954 FUTURE DIRECTIONS

19955 None.

19956 SEE ALSO

19957 *getsockopt()*, *if_freenameindex()*, *if_nameindex()*, *if_nametoindex()*, *setsockopt()*, the Base
19958 Definitions volume of IEEE Std 1003.1-2001, <net/if.h>

19959 CHANGE HISTORY

19960 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19961 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/28 is applied, changing {IFNAMSIZ} to 1
19962 {IF NAMESIZ} in the DESCRIPTION. 1

19963 NAME

19964 *if_nameindex* — return all network interface names and indexes

19965 SYNOPSIS

19966 #include <net/if.h>

19967 struct if_nameindex **if_nameindex*(void);

19968 DESCRIPTION

19969 The *if_nameindex()* function shall return an array of *if_nameindex* structures, one structure per
19970 interface. The end of the array is indicated by a structure with an *if_index* field of zero and an
19971 *if_name* field of NULL.

19972 Applications should call *if_freenameindex()* to release the memory that may be dynamically
19973 allocated by this function, after they have finished using it.

19974 RETURN VALUE

19975 An array of structures identifying local interfaces. A NULL pointer is returned upon an error,
19976 with *errno* set to indicate the error.

19977 ERRORS

19978 The *if_nameindex()* function may fail if:

19979 [ENOBUFS] Insufficient resources are available to complete the function.

19980 EXAMPLES

19981 None.

19982 APPLICATION USAGE

19983 None.

19984 RATIONALE

19985 None.

19986 FUTURE DIRECTIONS

19987 None.

19988 SEE ALSO

19989 *getsockopt()*, *if_freenameindex()*, *if_indextoname()*, *if_nametoindex()*, *setsockopt()*, the Base
19990 Definitions volume of IEEE Std 1003.1-2001, <net/if.h>

19991 CHANGE HISTORY

19992 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

19993 NAME

19994 **if_nametoindex** — map a network interface name to its corresponding index

19995 SYNOPSIS

19996 #include <net/if.h>

19997 unsigned if_nametoindex(const char **ifname*);

19998 DESCRIPTION

19999 The *if_nametoindex()* function shall return the interface index corresponding to name *ifname*.

20000 RETURN VALUE

20001 The corresponding index if *ifname* is the name of an interface; otherwise, zero.

20002 ERRORS

20003 No errors are defined.

20004 EXAMPLES

20005 None.

20006 APPLICATION USAGE

20007 None.

20008 RATIONALE

20009 None.

20010 FUTURE DIRECTIONS

20011 None.

20012 SEE ALSO

20013 *getsockopt()*, *if_freenameindex()*, *if_indextoname()*, *if_nameindex()*, *setsockopt()*, the Base
20014 Definitions volume of IEEE Std 1003.1-2001, <**net/if.h**>

20015 CHANGE HISTORY

20016 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

20017 NAME

20018 `ilogb`, `ilogbf`, `ilogbl` — return an unbiased exponent

20019 SYNOPSIS

```
20020        #include <math.h>
20021        int ilogb(double x);
20022        int ilogbf(float x);
20023        int ilogbl(long double x);
```

20024 DESCRIPTION

20025 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20026 conflict between the requirements described here and the ISO C standard is unintentional. This
20027 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20028 These functions shall return the exponent part of their argument x . Formally, the return value is
20029 the integral part of $\log_r |x|$ as a signed integral value, for non-zero x , where r is the radix of the
20030 machine's floating-point arithmetic, which is the value of `FLT_RADIX` defined in `<float.h>`.

20031 An application wishing to check for error situations should set `errno` to zero and call
20032 `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is non-zero or
20033 `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-
20034 zero, an error has occurred.

20035 RETURN VALUE

20036 Upon successful completion, these functions shall return the exponent part of x as a signed
20037 integer value. They are equivalent to calling the corresponding `logb()` function and casting the
20038 returned value to type `int`.

20039 XSI If x is 0, a domain error shall occur, and the value `FP_ILOGB0` shall be returned.

20040 XSI If x is $\pm\infty$, a domain error shall occur, and the value `{INT_MAX}` shall be returned.

20041 XSI If x is a NaN, a domain error shall occur, and the value `FP_ILOGBNAN` shall be returned.

20042 XSI If the correct value is greater than `{INT_MAX}`, `{INT_MAX}` shall be returned and a domain error
20043 shall occur.

20044 If the correct value is less than `{INT_MIN}`, `{INT_MIN}` shall be returned and a domain error
20045 shall occur.

20046 ERRORS

20047 These functions shall fail if:

20048 XSI Domain Error The x argument is zero, NaN, or $\pm\infty$, or the correct value is not representable
20049 as an integer.

20050 If the integer expression `(math_errhandling & MATH_ERRNO)` is non-zero,
20051 then `errno` shall be set to `[EDOM]`. If the integer expression `(math_errhandling`
20052 & `MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception
20053 shall be raised.

20054 EXAMPLES

20055 None.

20056 APPLICATION USAGE

20057 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

20059 RATIONALE

20060 The errors come from taking the expected floating-point value and converting it to **int**, which is
20061 an invalid operation in IEEE Std 754-1985 (since overflow, infinity, and NaN are not
20062 representable in a type **int**), so should be a domain error.

20063 There are no known implementations that overflow. For overflow to happen, **{INT_MAX}** must
20064 be less than **LDBL_MAX_EXP*log2(FLT_RADIX)** or **{INT_MIN}** must be greater than
20065 **LDBL_MIN_EXP*log2(FLT_RADIX)** if subnormals are not supported, or **{INT_MIN}** must be
20066 greater than **(LDBL_MIN_EXP-LDBL_MANT_DIG)*log2(FLT_RADIX)** if subnormals are
20067 supported.

20068 FUTURE DIRECTIONS

20069 None.

20070 SEE ALSO

20071 *feclearexcept()*, *fetestexcept()*, *logb()*, *scalb()*, the Base Definitions volume of IEEE Std 1003.1-2001,
20072 Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<float.h>**, **<math.h>**

20073 CHANGE HISTORY

20074 First released in Issue 4, Version 2.

20075 Issue 5

20076 Moved from X/OPEN UNIX extension to BASE.

20077 Issue 6

20078 The *ilogb()* function is no longer marked as an extension.

20079 The *ilogbf()* and *ilogbl()* functions are added for alignment with the ISO/IEC 9899:1999
20080 standard.

20081 The RETURN VALUE section is revised for alignment with the ISO/IEC 9899:1999 standard.

20082 XSI extensions are marked.

20083 NAME

20084 *imaxabs* — return absolute value

20085 SYNOPSIS

```
20086        #include <inttypes.h>
20087        intmax_t imaxabs(intmax_t j);
```

20088 DESCRIPTION

20089 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20090 conflict between the requirements described here and the ISO C standard is unintentional. This
20091 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20092 The *imaxabs()* function shall compute the absolute value of an integer *j*. If the result cannot be
20093 represented, the behavior is undefined.

20094 RETURN VALUE

20095 The *imaxabs()* function shall return the absolute value.

20096 ERRORS

20097 No errors are defined.

20098 EXAMPLES

20099 None.

20100 APPLICATION USAGE

20101 The absolute value of the most negative number cannot be represented in two's complement.

20102 RATIONALE

20103 None.

20104 FUTURE DIRECTIONS

20105 None.

20106 SEE ALSO

20107 *imaxdiv()*, the Base Definitions volume of IEEE Std 1003.1-2001, <inttypes.h>

20108 CHANGE HISTORY

20109 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

20110 NAME

20111 **imaxdiv** — return quotient and remainder

20112 SYNOPSIS

```
20113        #include <inttypes.h>
20114        imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

20115 DESCRIPTION

20116 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20117 conflict between the requirements described here and the ISO C standard is unintentional. This
20118 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20119 The *imaxdiv()* function shall compute *numer* / *denom* and *numer* % *denom* in a single operation.

20120 RETURN VALUE

20121 The *imaxdiv()* function shall return a structure of type **imaxdiv_t**, comprising both the quotient
20122 and the remainder. The structure shall contain (in either order) the members *quot* (the quotient)
20123 and *rem* (the remainder), each of which has type **intmax_t**.

20124 If either part of the result cannot be represented, the behavior is undefined.

20125 ERRORS

20126 No errors are defined.

20127 EXAMPLES

20128 None.

20129 APPLICATION USAGE

20130 None.

20131 RATIONALE

20132 None.

20133 FUTURE DIRECTIONS

20134 None.

20135 SEE ALSO

20136 *imaxabs()*, the Base Definitions volume of IEEE Std 1003.1-2001, <inttypes.h>

20137 CHANGE HISTORY

20138 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

20139 NAME

20140 index — character string operations (**LEGACY**)

20141 SYNOPSIS

20142 XSI `#include <strings.h>`

20143 `char *index(const char *s, int c);`

20144

20145 DESCRIPTION

20146 The *index()* function shall be equivalent to *strchr()*.

20147 RETURN VALUE

20148 See *strchr()*.

20149 ERRORS

20150 See *strchr()*.

20151 EXAMPLES

20152 None.

20153 APPLICATION USAGE

20154 The *strchr()* function is preferred over this function.

20155 For maximum portability, it is recommended to replace the function call to *index()* as follows:

20156 `#define index(a,b) strchr((a),(b))`

20157 RATIONALE

20158 None.

20159 FUTURE DIRECTIONS

20160 This function may be withdrawn in a future version.

20161 SEE ALSO

20162 *strchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<strings.h>**

20163 CHANGE HISTORY

20164 First released in Issue 4, Version 2.

20165 Issue 5

20166 Moved from X/OPEN UNIX extension to BASE.

20167 Issue 6

20168 This function is marked LEGACY.

20169 **NAME**

20170 inet_addr, inet_ntoa — IPv4 address manipulation

20171 **SYNOPSIS**

```
20172       #include <arpa/inet.h>
20173       in_addr_t inet_addr(const char *cp);
20174       char *inet_ntoa(struct in_addr in);
```

20175 **DESCRIPTION**

20176 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.

20178 The *inet_ntoa()* function shall convert the Internet host address specified by *in* to a string in the Internet standard dot notation.

20180 The *inet_ntoa()* function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

20182 All Internet addresses shall be returned in network order (bytes ordered from left to right).

20183 Values specified using IPv4 dotted decimal notation take one of the following forms:

20184 a.b.c.d When four parts are specified, each shall be interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

20186 a.b.c When a three-part address is specified, the last part shall be interpreted as a 16-bit quantity and placed in the rightmost two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as "128.net.host".

20190 a.b When a two-part address is supplied, the last part shall be interpreted as a 24-bit quantity and placed in the rightmost three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as "net.host".

20194 a When only one part is given, the value shall be stored directly in the network address without any byte rearrangement.

20196 All numbers supplied as parts in IPv4 dotted decimal notation may be decimal, octal, or hexadecimal, as specified in the ISO C standard (that is, a leading 0x or 0X implies hexadecimal; otherwise, a leading '0' implies octal; otherwise, the number is interpreted as decimal).

20199 **RETURN VALUE**

20200 Upon successful completion, *inet_addr()* shall return the Internet address. Otherwise, it shall return (*in_addr_t*)(-1).

20202 The *inet_ntoa()* function shall return a pointer to the network address in Internet standard dot notation.

20204 **ERRORS**

20205 No errors are defined.

20206 EXAMPLES

20207 None.

20208 APPLICATION USAGE

20209 The return value of *inet_ntoa()* may point to static data that may be overwritten by subsequent calls to *inet_ntoa()*.

20211 RATIONALE

20212 None.

20213 FUTURE DIRECTIONS

20214 None.

20215 SEE ALSO

20216 *endhostent()*, *endnetent()*, the Base Definitions volume of IEEE Std 1003.1-2001, <arpa/inet.h>

20217 CHANGE HISTORY

20218 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

20219 NAME

20220 inet_ntop, inet_pton — convert IPv4 and IPv6 addresses between binary and text form

20221 SYNOPSIS

```
20222     #include <arpa/inet.h>
20223
20224     const char *inet_ntop(int af, const void *restrict src,
20225         char *restrict dst, socklen_t size);
20226     int inet_pton(int af, const char *restrict src, void *restrict dst);
```

20226 DESCRIPTION

20227 The *inet_ntop()* function shall convert a numeric address into a text string suitable for
 20228 IP6 presentation. The *af* argument shall specify the family of the address. This can be AF_INET or
 20229 AF_INET6. The *src* argument points to a buffer holding an IPv4 address if the *af* argument is
 20230 IP6 AF_INET, or an IPv6 address if the *af* argument is AF_INET6; the address must be in network
 20231 byte order. The *dst* argument points to a buffer where the function stores the resulting text
 20232 string; it shall not be NULL. The *size* argument specifies the size of this buffer, which shall be
 20233 IP6 large enough to hold the text string (INET_ADDRSTRLEN characters for IPv4,
 20234 INET6_ADDRSTRLEN characters for IPv6).

20235 The *inet_pton()* function shall convert an address in its standard text presentation form into its
 20236 IP6 numeric binary form. The *af* argument shall specify the family of the address. The AF_INET and
 20237 AF_INET6 address families shall be supported. The *src* argument points to the string being
 20238 passed in. The *dst* argument points to a buffer into which the function stores the numeric
 20239 IP6 address; this shall be large enough to hold the numeric address (32 bits for AF_INET, 128 bits for
 20240 AF_INET6).

20241 If the *af* argument of *inet_pton()* is AF_INET, the *src* string shall be in the standard IPv4 dotted-
 20242 decimal form:

20243 ddd.ddd.ddd.ddd

20244 where "ddd" is a one to three digit decimal number between 0 and 255 (see *inet_addr()*). The
 20245 *inet_pton()* function does not accept other formats (such as the octal numbers, hexadecimal
 20246 numbers, and fewer than four numbers that *inet_addr()* accepts).

20247 IP6 If the *af* argument of *inet_pton()* is AF_INET6, the *src* string shall be in one of the following
 20248 standard IPv6 text forms:

- 20249 1. The preferred form is "x:x:x:x:x:x:x:x", where the 'x's are the hexadecimal values
 20250 of the eight 16-bit pieces of the address. Leading zeros in individual fields can be omitted,
 20251 but there shall be at least one numeral in every field.
- 20252 2. A string of contiguous zero fields in the preferred form can be shown as ":::". The ":"
 20253 can only appear once in an address. Unspecified addresses ("0:0:0:0:0:0:0:0") may
 20254 be represented simply as "::".
- 20255 3. A third form that is sometimes more convenient when dealing with a mixed environment
 20256 of IPv4 and IPv6 nodes is "x:x:x:x:x:d.d.d.d", where the 'x's are the
 20257 hexadecimal values of the six high-order 16-bit pieces of the address, and the 'd's are the
 20258 decimal values of the four low-order 8-bit pieces of the address (standard IPv4
 20259 representation).

20260 **Note:** A more extensive description of the standard representations of IPv6 addresses can be found in
 20261 RFC 2373.

20262

20263 **RETURN VALUE**

20264 The *inet_ntop()* function shall return a pointer to the buffer containing the text string if the
20265 conversion succeeds, and NULL otherwise, and set *errno* to indicate the error.

20266 The *inet_pton()* function shall return 1 if the conversion succeeds, with the address pointed to by
20267 *dst* in network byte order. It shall return 0 if the input is not a valid IPv4 dotted-decimal string or
20268 a valid IPv6 address string, or -1 with *errno* set to [EAFNOSUPPORT] if the *af* argument is
20269 unknown.

20270 **ERRORS**

20271 The *inet_ntop()* and *inet_pton()* functions shall fail if:

20272 [EAFNOSUPPORT]

20273 The *af* argument is invalid.

20274 [ENOSPC] The size of the *inet_ntop()* result buffer is inadequate.

20275 **EXAMPLES**

20276 None.

20277 **APPLICATION USAGE**

20278 None.

20279 **RATIONALE**

20280 None.

20281 **FUTURE DIRECTIONS**

20282 None.

20283 **SEE ALSO**

20284 The Base Definitions volume of IEEE Std 1003.1-2001, <arpa/inet.h>

20285 **CHANGE HISTORY**

20286 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

20287 IPv6 extensions are marked.

20288 The **restrict** keyword is added to the *inet_ntop()* and *inet_pton()* prototypes for alignment with
20289 the ISO/IEC 9899:1999 standard.

20290 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/29 is applied, adding “the address must
20291 be in network byte order” to the end of the fourth sentence of the first paragraph in the
20292 DESCRIPTION. 1 1 1

20293 NAME

20294 initstate, random, setstate, srand — pseudo-random number functions

20295 SYNOPSIS

```
20296 XSI #include <stdlib.h>
20297     char *initstate(unsigned seed, char *state, size_t size);
20298     long random(void);
20299     char *setstate(const char *state);
20300     void srand(unsigned seed);
20301
```

20302 DESCRIPTION

20303 The *random()* function shall use a non-linear additive feedback random-number generator
20304 employing a default state array size of 31 **long** integers to return successive pseudo-random
20305 numbers in the range from 0 to $2^{31}-1$. The period of this random-number generator is
20306 approximately $16 \times (2^{31}-1)$. The size of the state array determines the period of the random-
20307 number generator. Increasing the state array size shall increase the period.

20308 With 256 bytes of state information, the period of the random-number generator shall be greater
20309 than 2^{69} .

20310 Like *rand()*, *random()* shall produce by default a sequence of numbers that can be duplicated by
20311 calling *srand()* with 1 as the seed.

20312 The *srand()* function shall initialize the current state array using the value of *seed*.

20313 The *initstate()* and *setstate()* functions handle restarting and changing random-number
20314 generators. The *initstate()* function allows a state array, pointed to by the *state* argument, to be
20315 initialized for future use. The *size* argument, which specifies the size in bytes of the state array,
20316 shall be used by *initstate()* to decide what type of random-number generator to use; the larger
20317 the state array, the more random the numbers. Values for the amount of state information are 8,
20318 32, 64, 128, and 256 bytes. Other values greater than 8 bytes are rounded down to the nearest one
20319 of these values. If *initstate()* is called with $8 \leq \text{size} \leq 32$, then *random()* shall use a simple linear
20320 congruential random number generator. The *seed* argument specifies a starting point for the
20321 random-number sequence and provides for restarting at the same point. The *initstate()* function
20322 shall return a pointer to the previous state information array.

20323 If *initstate()* has not been called, then *random()* shall behave as though *initstate()* had been called
20324 with *seed*=1 and *size*=128.

20325 Once a state has been initialized, *setstate()* allows switching between state arrays. The array
20326 defined by the *state* argument shall be used for further random-number generation until
20327 *initstate()* is called or *setstate()* is called again. The *setstate()* function shall return a pointer to the
20328 previous state array.

20329 RETURN VALUE

20330 If *initstate()* is called with *size* less than 8, it shall return NULL.

20331 The *random()* function shall return the generated pseudo-random number.

20332 The *srand()* function shall not return a value.

20333 Upon successful completion, *initstate()* and *setstate()* shall return a pointer to the previous state
20334 array; otherwise, a null pointer shall be returned.

20335 ERRORS

20336 No errors are defined.

20337 EXAMPLES

20338 None.

20339 APPLICATION USAGE

20340 After initialization, a state array can be restarted at a different point in one of two ways:

- 20341 1. The *initstate()* function can be used, with the desired seed, state array, and size of the array.
- 20342 2. The *setstate()* function, with the desired state, can be used, followed by *random()* with the desired seed. The advantage of using both of these functions is that the size of the state array does not have to be saved once it is initialized.

20343 Although some implementations of *random()* have written messages to standard error, such
20344 implementations do not conform to IEEE Std 1003.1-2001.
20345

20346 Issue 5 restored the historical behavior of this function.
20347

20348 Threaded applications should use *erand48()*, *nrand48()*, or *jrand48()* instead of *random()* when 1
20349 an independent random number sequence in multiple threads is required.
20350

20351 RATIONALE

20352 None.

20353 FUTURE DIRECTIONS

20354 None.

20355 SEE ALSO

20356 *drand48()*, *rand()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

20357 CHANGE HISTORY

20358 First released in Issue 4, Version 2.

20359 Issue 5

20360 Moved from X/OPEN UNIX extension to BASE.

20361 In the DESCRIPTION, the phrase “values smaller than 8” is replaced with “values greater than
20362 or equal to 8, or less than 32”, “size<8” is replaced with “8≤size <32”, and a new first paragraph
20363 is added to the RETURN VALUE section. A note is added to the APPLICATION USAGE
20364 indicating that these changes restore the historical behavior of the function.

20365 Issue 6

20366 In the DESCRIPTION, duplicate text “For values greater than or equal to 8 . . .” is removed.

20367 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/30 is applied, removing *rand_r()* from the 1
20368 list of suggested functions in the APPLICATION USAGE section. 1

20369 NAME

20370 insque, remque — insert or remove an element in a queue

20371 SYNOPSIS

20372 XSI #include <search.h>

20373 void insque(void *element, void *pred);

20374 void remque(void *element);

20375

20376 DESCRIPTION

20377 The *insque()* and *remque()* functions shall manipulate queues built from doubly-linked lists. The queue can be either circular or linear. An application using *insque()* or *remque()* shall ensure it defines a structure in which the first two members of the structure are pointers to the same type of structure, and any further members are application-specific. The first member of the structure is a forward pointer to the next entry in the queue. The second member is a backward pointer to the previous entry in the queue. If the queue is linear, the queue is terminated with null pointers. The names of the structure and of the pointer members are not subject to any special restriction.

20385 The *insque()* function shall insert the element pointed to by *element* into a queue immediately after the element pointed to by *pred*.

20387 The *remque()* function shall remove the element pointed to by *element* from a queue.

20388 If the queue is to be used as a linear list, invoking *insque(&element, NULL)*, where *element* is the initial element of the queue, shall initialize the forward and backward pointers of *element* to null pointers.

20391 If the queue is to be used as a circular list, the application shall ensure it initializes the forward pointer and the backward pointer of the initial element of the queue to the element's own address.

20394 RETURN VALUE

20395 The *insque()* and *remque()* functions do not return a value.

20396 ERRORS

20397 No errors are defined.

20398 EXAMPLES

20399 Creating a Linear Linked List

20400 The following example creates a linear linked list.

```
20401 #include <search.h>
20402 ...
20403 struct myque element1;
20404 struct myque element2;
20405 char *data1 = "DATA1";
20406 char *data2 = "DATA2";
20407 ...
20408 element1.data = data1;
20409 element2.data = data2;
20410 insque (&element1, NULL);
20411 insque (&element2, &element1);
```

20412 **Creating a Circular Linked List**

20413 The following example creates a circular linked list.

```
20414       #include <search.h>
20415       ...
20416       struct myque element1;
20417       struct myque element2;
20418       char *data1 = "DATA1";
20419       char *data2 = "DATA2";
20420       ...
20421       element1.data = data1;
20422       element2.data = data2;
20423       element1.fwd = &element1;
20424       element1.bck = &element1;
20425       insque (&element2, &element1);
```

20426 **Removing an Element**

20427 The following example removes the element pointed to by *element1*.

```
20428       #include <search.h>
20429       ...
20430       struct myque element1;
20431       ...
20432       remque (&element1);
```

20433 **APPLICATION USAGE**

20434 The historical implementations of these functions described the arguments as being of type
20435 **struct qelem *** rather than as being of type **void *** as defined here. In those implementations,
20436 **struct qelem** was commonly defined in **<search.h>** as:

```
20437       struct qelem {
20438            struct qelem *q_forw;
20439            struct qelem *q_back;
20440       };
```

20441 Applications using these functions, however, were never able to use this structure directly since
20442 it provided no room for the actual data contained in the elements. Most applications defined
20443 structures that contained the two pointers as the initial elements and also provided space for, or
20444 pointers to, the object's data. Applications that used these functions to update more than one
20445 type of table also had the problem of specifying two or more different structures with the same
20446 name, if they literally used **struct qelem** as specified.

20447 As described here, the implementations were actually expecting a structure type where the first
20448 two members were forward and backward pointers to structures. With C compilers that didn't
20449 provide function prototypes, applications used structures as specified in the DESCRIPTION
20450 above and the compiler did what the application expected.

20451 If this method had been carried forward with an ISO C standard compiler and the historical
20452 function prototype, most applications would have to be modified to cast pointers to the
20453 structures actually used to be pointers to **struct qelem** to avoid compilation warnings. By
20454 specifying **void *** as the argument type, applications do not need to change (unless they
20455 specifically referenced **struct qelem** and depended on it being defined in **<search.h>**).

20456 RATIONALE

20457 None.

20458 FUTURE DIRECTIONS

20459 None.

20460 SEE ALSO

20461 The Base Definitions volume of IEEE Std 1003.1-2001, <search.h>

20462 CHANGE HISTORY

20463 First released in Issue 4, Version 2.

20464 Issue 5

20465 Moved from X/OPEN UNIX extension to BASE.

20466 Issue 6

20467 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

20468 NAME

20469 ioctl — control a STREAMS device (**STREAMS**)

20470 SYNOPSIS

```
20471 XSR #include <stropts.h>
20472     int ioctl(int fildes, int request, ... /* arg */);
```

20474 DESCRIPTION

20475 The *ioctl()* function shall perform a variety of control functions on STREAMS devices. For non-
20476 STREAMS devices, the functions performed by this call are unspecified. The *request* argument
20477 and an optional third argument (with varying type) shall be passed to and interpreted by the
20478 appropriate part of the STREAM associated with *fildes*.

20479 The *fildes* argument is an open file descriptor that refers to a device.

20480 The *request* argument selects the control function to be performed and shall depend on the
20481 STREAMS device being addressed.

20482 The *arg* argument represents additional information that is needed by this specific STREAMS
20483 device to perform the requested function. The type of *arg* depends upon the particular control
20484 request, but it shall be either an integer or a pointer to a device-specific data structure.

20485 The *ioctl()* commands applicable to STREAMS, their arguments, and error conditions that apply
20486 to each individual command are described below.

20487 The following *ioctl()* commands, with error values indicated, are applicable to all STREAMS
20488 files:

20489 **I_PUSH** Pushes the module whose name is pointed to by *arg* onto the top of the
20490 current STREAM, just below the STREAM head. It then calls the *open()*
20491 function of the newly-pushed module.

20492 The *ioctl()* function with the **I_PUSH** command shall fail if:

20493 [EINVAL] Invalid module name.

20494 [ENXIO] Open function of new module failed.

20495 [ENXIO] Hangup received on *fildes*.

20496 **I_POP** Removes the module just below the STREAM head of the STREAM pointed to
20497 by *fildes*. The *arg* argument should be 0 in an **I_POP** request.

20498 The *ioctl()* function with the **I_POP** command shall fail if:

20499 [EINVAL] No module present in the STREAM.

20500 [ENXIO] Hangup received on *fildes*.

20501 **I_LOOK** Retrieves the name of the module just below the STREAM head of the
20502 STREAM pointed to by *fildes*, and places it in a character string pointed to by
20503 *arg*. The buffer pointed to by *arg* should be at least FMNAMESZ+1 bytes long,
20504 where FMNAMESZ is defined in **<stropts.h>**.

20505 The *ioctl()* function with the **I_LOOK** command shall fail if:

20506 [EINVAL] No module present in the STREAM.

20507 **I_FLUSH** Flushes read and/or write queues, depending on the value of *arg*. Valid *arg*
20508 values are:

20509	FLUSHR	Flush all read queues.
20510	FLUSHW	Flush all write queues.
20511	FLUSHRW	Flush all read and all write queues.
20512		The <i>ioctl()</i> function with the I_FLUSH command shall fail if:
20513	[EINVAL]	Invalid <i>arg</i> value.
20514	[EAGAIN] or [ENOSR]	
20515		Unable to allocate buffers for flush message.
20516	[ENXIO]	Hangup received on <i>fildes</i> .
20517	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.
20521	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>fildes</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:
20527	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.
20530	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.
20533	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.
20536	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.
20539	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.
20543	S_WRNORM	Equivalent to S_OUTPUT.
20544	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.
20548	S_MSG	A STREAMS signal message that contains the SIGPOLL signal has reached the front of the STREAM head read queue.
20551	S_ERROR	Notification of an error condition has reached the STREAM head.

20553	S_HANGUP	Notification of a hangup has reached the STREAM head.
20554	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.
20555		
20556		
20557		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>fd</i> .
20558		
20559		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.
20560		
20561		
20562		
20563		The <i>ioctl()</i> function with the I_SETSIG command shall fail if:
20564	[EINVAL]	The value of <i>arg</i> is invalid.
20565	[EINVAL]	The value of <i>arg</i> is 0 and the calling process is not registered to receive the SIGPOLL signal.
20566		
20567	[EAGAIN]	There were insufficient resources to store the signal request.
20568	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 int pointed to by <i>arg</i> , where the events are those specified in the description of I_SETSIG above.
20569		
20570		
20571		
20572		The <i>ioctl()</i> function with the I_GETSIG command shall fail if:
20573	[EINVAL]	Process is not registered to receive the SIGPOLL signal.
20574	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.
20575		
20576		
20577		The <i>ioctl()</i> function with the I_FIND command shall fail if:
20578	[EINVAL]	<i>arg</i> does not contain a valid module name.
20579	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.
20580		
20581		
20582		
20583		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.
20584		
20585		
20586		
20587		
20588		
20589		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.
20590		
20591		
20592		
20593		
20594		
20595	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:
20596		

20597	RNORM	Byte-stream mode, the default.
20598	RMSGD	Message-discard mode.
20599	RMSGN	Message-nondiscard mode.
20600		The bitwise-inclusive OR of RMSGD and RMSGN shall return [EINVAL]. The
20601		bitwise-inclusive OR of RNORM and either RMSGD or RMSGN shall result in
20602		the other flag overriding RNORM which is the default.
20603		In addition, treatment of control messages by the STREAM head may be
20604		changed by setting any of the following flags in <i>arg</i> :
20605	RPROTNORM	Fail <i>read()</i> with [EBADMSG] if a message containing a
20606		control part is at the front of the STREAM head read queue.
20607	RPROTDAT	Deliver the control part of a message as data when a
20608		process issues a <i>read()</i> .
20609	RPROTDIS	Discard the control part of a message, delivering any data
20610		portion, when a process issues a <i>read()</i> .
20611		The <i>ioctl()</i> function with the I_SRDOPT command shall fail if:
20612	[EINVAL]	The <i>arg</i> argument is not valid.
20613	I_GRDOPT	Returns the current read mode setting, as described above, in an int pointed to
20614		by the argument <i>arg</i> . Read modes are described in <i>read()</i> .
20615	I_NREAD	Counts the number of data bytes in the data part of the first message on the
20616		STREAM head read queue and places this value in the int pointed to by <i>arg</i> .
20617		The return value for the command shall be the number of messages on the
20618		STREAM head read queue. For example, if 0 is returned in <i>arg</i> , but the <i>ioctl()</i>
20619		return value is greater than 0, this indicates that a zero-length message is next
20620		on the queue.
20621	I_FDINSERT	Creates a message from specified buffer(s), adds information about another
20622		STREAM, and sends the message downstream. The message contains a
20623		control part and an optional data part. The data and control parts to be sent
20624		are distinguished by placement in separate buffers, as described below. The
20625		<i>arg</i> argument points to a strfdinsert structure.
20626		The application shall ensure that the <i>len</i> member in the ctlbuf strbuf structure
20627		is set to the size of a t_uscalar_t plus the number of bytes of control
20628		information to be sent with the message. The <i>fildes</i> member specifies the file
20629		descriptor of the other STREAM, and the <i>offset</i> member, which must be
20630		suitably aligned for use as a t_uscalar_t , specifies the offset from the start of
20631		the control buffer where I_FDINSERT shall store a t_uscalar_t whose
20632		interpretation is specific to the STREAM end. The application shall ensure that
20633		the <i>len</i> member in the databuf strbuf structure is set to the number of bytes of
20634		data information to be sent with the message, or to 0 if no data part is to be
20635		sent.
20636		The <i>flags</i> member specifies the type of message to be created. A normal
20637		message is created if <i>flags</i> is set to 0, and a high-priority message is created if
20638		<i>flags</i> is set to RS_HIPRI. For non-priority messages, I_FDINSERT shall block if
20639		the STREAM write queue is full due to internal flow control conditions. For
20640		priority messages, I_FDINSERT does not block on this condition. For non-
20641		priority messages, I_FDINSERT does not block when the write queue is full

20642		and O_NONBLOCK is set. Instead, it fails and sets <i>errno</i> to [EAGAIN].
20643		I_FDINSERT also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the STREAM, regardless of priority or whether O_NONBLOCK has been specified. No partial message is sent.
20644		
20645		
20646		
20647		The <i>ioctl()</i> function with the I_FDINSERT command shall fail if:
20648	[EAGAIN]	A non-priority message is specified, the O_NONBLOCK flag is set, and the STREAM write queue is full due to internal flow control conditions.
20649		
20650		
20651	[EAGAIN] or [ENOSR]	
20652		Buffers cannot be allocated for the message that is to be created.
20653		
20654	[EINVAL]	One of the following:
20655		— The <i>fd</i> member of the strfdinsert structure is not a valid, open STREAM file descriptor.
20656		
20657		— The size of a t_ustcalar_t plus <i>offset</i> is greater than the <i>len</i> member for the buffer specified through <i>ctlbuf</i> .
20658		
20659		— The <i>offset</i> member does not specify a properly-aligned location in the data buffer.
20660		
20661		— An undefined value is stored in <i>flags</i> .
20662	[ENXIO]	Hangup received on the STREAM identified by either the <i>fd</i> argument or the <i>fd</i> member of the strfdinsert structure.
20663		
20664		
20665	[ERANGE]	The <i>len</i> member for the buffer specified through databuf does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module; or the <i>len</i> member for the buffer specified through databuf is larger than the maximum configured size of the data part of a message; or the <i>len</i> member for the buffer specified through <i>ctlbuf</i> is larger than the maximum configured size of the control part of a message.
20666		
20667		
20668		
20669		
20670		
20671		
20672		
20673	I_STR	Constructs an internal STREAMS <i>ioctl()</i> message from the data pointed to by <i>arg</i> , and sends that message downstream.
20674		
20675		This mechanism is provided to send <i>ioctl()</i> requests to downstream modules and drivers. It allows information to be sent with <i>ioctl()</i> , and returns to the process any information sent upstream by the downstream recipient. I_STR shall block until the system responds with either a positive or negative acknowledgement message, or until the request times out after some period of time. If the request times out, it shall fail with <i>errno</i> set to [ETIME].
20676		
20677		
20678		
20679		
20680		
20681		
20682		
20683		
20684		
20685		At most, one I_STR can be active on a STREAM. Further I_STR calls shall block until the active I_STR completes at the STREAM head. The default timeout interval for these requests is 15 seconds. The O_NONBLOCK flag has no effect on this call.
20686		
20685		To send requests downstream, the application shall ensure that <i>arg</i> points to a stroctl structure.
20686		

20687
 20688
 20689
 20690
 20691
 20692
 20693
 20694
 20695
 20696

The *ic_cmd* member is the internal *ioctl()* command intended for a downstream module or driver and *ic_timeout* 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. *ic_len* is the number of bytes in the data argument, and *ic_dp* is a pointer to the data argument. The *ic_len* 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 *ic_dp* should be large enough to contain the maximum amount of data that any module or the driver in the STREAM can return).

20697
 20698

The STREAM head shall convert the information pointed to by the **stroctl** structure to an internal *ioctl()* command message and send it downstream.

20699

The *ioctl()* function with the I_STR command shall fail if:

20700
 20701

[EAGAIN] or [ENOSR]	Unable to allocate buffers for the <i>ioctl()</i> message.
---------------------	--

20702
 20703
 20704

[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.
----------	---

20705

[ENXIO]	Hangup received on <i>fd</i> .
---------	--------------------------------

20706
 20707

[ETIME]	A downstream <i>ioctl()</i> timed out before acknowledgement was received.
---------	--

20708
 20709
 20710
 20711
 20712

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 *ioctl()* command sent downstream fails. For these cases, I_STR shall fail with *errno* set to the value in the message.

20713 I_SWROPT

Sets the write mode using the value of the argument *arg*. Valid bit settings for *arg* are:

20714

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).
---------	--

20715
 20716
 20717
 20718

20719

The *ioctl()* function with the I_SWROPT command shall fail if:

20720

[EINVAL]	<i>arg</i> is not the above value.
----------	------------------------------------

20721 I_GWROPT

Returns the current write mode setting, as described above, in the *int* that is pointed to by the argument *arg*.

20722

20723 I_SENDFD

Creates a new reference to the open file description associated with the file descriptor *arg*, and writes a message on the STREAMS-based pipe *fd* containing this reference, together with the user ID and group ID of the calling process.

20724
 20725
 20726

20727

The *ioctl()* function with the I_SENDFD command shall fail if:

20728
 20729
 20730
 20731

[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.
----------	--

20732	[EBADF]	The <i>arg</i> argument is not a valid, open file descriptor.
20733	[EINVAL]	The <i>fildes</i> argument is not connected to a STREAM pipe.
20734	[ENXIO]	Hangup received on <i>fildes</i> .
20735	I_RECVFD	<p>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>.</p> <p>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.</p> <p>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.</p> <p>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>.</p>
20740		The <i>ioctl()</i> function with the I_RECVFD command shall fail if:
20741	[EAGAIN]	A message is not present at the STREAM head read queue and the O_NONBLOCK flag is set.
20742	[EBADMSG]	The message at the STREAM head read queue is not a message containing a passed file descriptor.
20743	[EMFILE]	The process has the maximum number of file descriptors currently open that it is allowed.
20744	[ENXIO]	Hangup received on <i>fildes</i> .
20745	I_LIST	<p>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>fildes</i>. This lets the process allocate enough space for the module names. Otherwise, it should point to a str_list structure.</p> <p>The <i>sl_nmmods</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_nmmods</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_nmmods</i>) is satisfied.</p>
20750		The <i>ioctl()</i> function with the I_LIST command shall fail if:
20751	[EINVAL]	The <i>sl_nmmods</i> member is less than 1.
20752	[EAGAIN] or [ENOSR]	Unable to allocate buffers.
20753		
20754	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

20776		how the checking is done when there may be multiple marked messages on the STREAM head read queue. It may take on the following values:
20777		
20778	ANYMARK	Check if the message is marked.
20779	LASTMARK	Check if the message is the last one marked on the queue.
20780		The bitwise-inclusive OR of the flags ANYMARK and LASTMARK is permitted.
20781		
20782		The return value shall be 1 if the mark condition is satisfied; otherwise, the value shall be 0.
20783		
20784		The <i>ioctl()</i> function with the I_ATMMARK command shall fail if:
20785	[EINVAL]	Invalid <i>arg</i> value.
20786	I_CKBAND	Checks if the message of a given priority band exists on the STREAM head read queue. This shall return 1 if a message of the given priority exists, 0 if no such message exists, or -1 on error. <i>arg</i> should be of type int .
20787		
20788		The <i>ioctl()</i> function with the I_CKBAND command shall fail if:
20789	[EINVAL]	Invalid <i>arg</i> value.
20790	I_GETBAND	Returns the priority band of the first message on the STREAM head read queue in the integer referenced by <i>arg</i> .
20791		The <i>ioctl()</i> function with the I_GETBAND command shall fail if:
20792	[ENODATA]	No message on the STREAM head read queue.
20793		
20794	I_CANPUT	Checks if a certain band is writable. <i>arg</i> is set to the priority band in question. The return value shall be 0 if the band is flow-controlled, 1 if the band is writable, or -1 on error.
20795		The <i>ioctl()</i> function with the I_CANPUT command shall fail if:
20796	[EINVAL]	Invalid <i>arg</i> value.
20797		
20798	I_SETCLTIME	This request allows the process to set the time the STREAM head shall delay when a STREAM is closing and there is data on the write queues. Before closing each module or driver, if there is data on its write queue, the STREAM head shall delay for the specified amount of time to allow the data to drain. If, after the delay, data is still present, it shall be flushed. The <i>arg</i> argument is a pointer to an integer specifying the number of milliseconds to delay, rounded up to the nearest valid value. If I_SETCLTIME is not performed on a STREAM, an implementation-defined default timeout interval is used.
20799		
20800		The <i>ioctl()</i> function with the I_SETCLTIME command shall fail if:
20801	[EINVAL]	Invalid <i>arg</i> value.
20802		
20803		
20804		
20805		
20806		
20807		
20808		
20809		
20810	I_GETCLTIME	Returns the close time delay in the integer pointed to by <i>arg</i> .
20811		

20811	Multiplexed STREAMS Configurations
20812	The following commands are used for connecting and disconnecting multiplexed STREAMS configurations. These commands use an implementation-defined default timeout interval.
20814	I_LINK Connects two STREAMs, where <i>fdes</i> is the file descriptor of the STREAM connected to the multiplexing driver, and <i>arg</i> is the file descriptor of the STREAM connected to another driver. The STREAM designated by <i>arg</i> is connected below the multiplexing driver. I_LINK requires the multiplexing driver to send an acknowledgement message to the STREAM head regarding the connection. This call shall return a multiplexer ID number (an identifier used to disconnect the multiplexer; see I_UNLINK) on success, and -1 on failure.
20822	The <i>ioctl()</i> function with the I_LINK command shall fail if:
20823	[ENXIO] Hangup received on <i>fdes</i> .
20824	[ETIME] Timeout before acknowledgement message was received at STREAM head.
20826	[EAGAIN] or [ENOSR]
20827	Unable to allocate STREAMS storage to perform the I_LINK .
20829	[EBADF] The <i>arg</i> argument is not a valid, open file descriptor.
20830	[EINVAL]
20831	The <i>fdes</i> argument does not support multiplexing; or <i>arg</i> is not a STREAM or is already connected downstream from a multiplexer; or the specified I_LINK operation would connect the STREAM head in more than one place in the multiplexed STREAM.
20835	An I_LINK can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hangup is received at the STREAM head of <i>fdes</i> . In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_LINK fails with <i>errno</i> set to the value in the message.
20840	I_UNLINK Disconnects the two STREAMs specified by <i>fdes</i> and <i>arg</i> . <i>fdes</i> is the file descriptor of the STREAM connected to the multiplexing driver. The <i>arg</i> argument is the multiplexer ID number that was returned by the I_LINK <i>ioctl()</i> command when a STREAM was connected downstream from the multiplexing driver. If <i>arg</i> is MUXID_ALL, then all STREAMs that were connected to <i>fdes</i> shall be disconnected. As in I_LINK , this command requires acknowledgement.
20847	The <i>ioctl()</i> function with the I_UNLINK command shall fail if:
20848	[ENXIO] Hangup received on <i>fdes</i> .
20849	[ETIME] Timeout before acknowledgement message was received at STREAM head.
20851	[EAGAIN] or [ENOSR]
20852	Unable to allocate buffers for the acknowledgement message.
20854	[EINVAL] Invalid multiplexer ID number.

20855		An I_UNLINK can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hangup is received at the STREAM head of <i>fdles</i> . In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_UNLINK shall fail with <i>errno</i> set to the value in the message.
20860	I_PLINK	Creates a <i>persistent connection</i> between two STREAMS, where <i>fdles</i> is the file descriptor of the STREAM connected to the multiplexing driver, and <i>arg</i> is the file descriptor of the STREAM connected to another driver. This call shall create a persistent connection which can exist even if the file descriptor <i>fdles</i> associated with the upper STREAM to the multiplexing driver is closed. The STREAM designated by <i>arg</i> gets connected via a persistent connection below the multiplexing driver. I_PLINK requires the multiplexing driver to send an acknowledgement message to the STREAM head. This call shall return a multiplexer ID number (an identifier that may be used to disconnect the multiplexer; see I_PUNLINK) on success, and -1 on failure.
20870		The <i>ioctl()</i> function with the I_PLINK command shall fail if:
20871	[ENXIO]	Hangup received on <i>fdles</i> .
20872	[ETIME]	Timeout before acknowledgement message was received at STREAM head.
20873		
20874	[EAGAIN] or [ENOSR]	Unable to allocate STREAMS storage to perform the I_PLINK.
20875		
20876	[EBADF]	The <i>arg</i> argument is not a valid, open file descriptor.
20877		
20878	[EINVAL]	The <i>fdles</i> argument does not support multiplexing; or <i>arg</i> is not a STREAM or is already connected downstream from a multiplexer; or the specified I_PLINK operation would connect the STREAM head in more than one place in the multiplexed STREAM.
20879		
20880		
20881		
20882		
20883		An I_PLINK can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hangup is received at the STREAM head of <i>fdles</i> . In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_PLINK shall fail with <i>errno</i> set to the value in the message.
20884		
20885		
20886		
20887		
20888	I_PUNLINK	Disconnects the two STREAMs specified by <i>fdles</i> and <i>arg</i> from a persistent connection. The <i>fdles</i> argument is the file descriptor of the STREAM connected to the multiplexing driver. The <i>arg</i> argument is the multiplexer ID number that was returned by the I_PLINK <i>ioctl()</i> command when a STREAM was connected downstream from the multiplexing driver. If <i>arg</i> is MUXID_ALL, then all STREAMs which are persistent connections to <i>fdles</i> shall be disconnected. As in I_PLINK, this command requires the multiplexing driver to acknowledge the request.
20889		
20890		
20891		
20892		
20893		
20894		
20895		
20896		The <i>ioctl()</i> function with the I_PUNLINK command shall fail if:
20897	[ENXIO]	Hangup received on <i>fdles</i> .
20898	[ETIME]	Timeout before acknowledgement message was received at STREAM head.
20899		

20900 [EAGAIN] or [ENOSR]
20901 Unable to allocate buffers for the acknowledgement
20902 message.
20903 [EINVAL] Invalid multiplexer ID number.
20904 An I_PUNLINK can also fail while waiting for the multiplexing driver to
20905 acknowledge the request if a message indicating an error or a hangup is
20906 received at the STREAM head of *fdes*. In addition, an error code can be
20907 returned in the positive or negative acknowledgement message. For these
20908 cases, I_PUNLINK shall fail with *errno* set to the value in the message.

20909 RETURN VALUE

20910 Upon successful completion, *ioctl()* shall return a value other than -1 that depends upon the
20911 STREAMS device control function. Otherwise, it shall return -1 and set *errno* to indicate the
20912 error.

20913 ERRORS

20914 Under the following general conditions, *ioctl()* shall fail if:
20915 [EBADF] The *fdes* argument is not a valid open file descriptor.
20916 [EINTR] A signal was caught during the *ioctl()* operation.
20917 [EINVAL] The STREAM or multiplexer referenced by *fdes* is linked (directly or
20918 indirectly) downstream from a multiplexer.
20919 If an underlying device driver detects an error, then *ioctl()* shall fail if:
20920 [EINVAL] The *request* or *arg* argument is not valid for this device.
20921 [EIO] Some physical I/O error has occurred.
20922 [ENOTTY] The *fdes* argument is not associated with a STREAMS device that accepts
20923 control functions.
20924 [ENXIO] The *request* and *arg* arguments are valid for this device driver, but the service
20925 requested cannot be performed on this particular sub-device.
20926 [ENODEV] The *fdes* argument refers to a valid STREAMS device, but the corresponding
20927 device driver does not support the *ioctl()* function.
20928 If a STREAM is connected downstream from a multiplexer, any *ioctl()* command except
20929 I_UNLINK and I_PUNLINK shall set *errno* to [EINVAL].

20930 EXAMPLES

20931 None.

20932 APPLICATION USAGE

20933 The implementation-defined timeout interval for STREAMS has historically been 15 seconds.

20934 RATIONALE

20935 None.

20936 FUTURE DIRECTIONS

20937 None.

20938 SEE ALSO

20939 Section 2.6 (on page 38), *close()*, *fcntl()*, *getmsg()*, *open()*, *pipe()*, *poll()*, *putmsg()*, *read()*,
20940 *sigaction()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stropts.h>

20941 CHANGE HISTORY

20942 First released in Issue 4, Version 2.

20943 Issue 5

20944 Moved from X/OPEN UNIX extension to BASE.

20945 Issue 6

20946 The Open Group Corrigendum U028/4 is applied, correcting text in the I_FDINSERT [EINVAL] case to refer to *ctlbuf*.

20948 This function is marked as part of the XSI STREAMS Option Group.

20949 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

20950 NAME

20951 isalnum — test for an alphanumeric character

20952 SYNOPSIS

```
20953        #include <ctype.h>
20954        int isalnum(int c);
```

20955 DESCRIPTION

20956 CX The functionality described on this reference page is aligned with the ISO C standard. Any
20957 conflict between the requirements described here and the ISO C standard is unintentional. This
20958 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

20959 The *isalnum()* function shall test whether *c* is a character of class **alpha** or **digit** in the program's
20960 current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

20961 The *c* argument is an **int**, the value of which the application shall ensure is representable as an
20962 **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the
20963 behavior is undefined.

20964 RETURN VALUE

20965 The *isalnum()* function shall return non-zero if *c* is an alphanumeric character; otherwise, it shall
20966 return 0.

20967 ERRORS

20968 No errors are defined.

20969 EXAMPLES

20970 None.

20971 APPLICATION USAGE

20972 To ensure applications portability, especially across natural languages, only this function and
20973 those listed in the SEE ALSO section should be used for character classification.

20974 RATIONALE

20975 None.

20976 FUTURE DIRECTIONS

20977 None.

20978 SEE ALSO

20979 *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*, *isxdigit()*,
20980 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, **<ctype.h>**,
20981 **<stdio.h>**

20982 CHANGE HISTORY

20983 First released in Issue 1. Derived from Issue 1 of the SVID.

20984 Issue 6

20985 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

20986 NAME

20987 *isalpha* — test for an alphabetic character

20988 SYNOPSIS

20989 #include <ctype.h>
20990 int isalpha(int c);

20991 DESCRIPTION

20992 CX 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.

20995 The *isalpha()* function shall test whether *c* is a character of class **alpha** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

20997 The *c* argument is an **int**, the value of which the application shall ensure is representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

21000 RETURN VALUE

21001 The *isalpha()* function shall return non-zero if *c* is an alphabetic character; otherwise, it shall return 0.

21003 ERRORS

21004 No errors are defined.

21005 EXAMPLES

21006 None.

21007 APPLICATION USAGE

21008 To ensure applications portability, especially across natural languages, only this function and
21009 those listed in the SEE ALSO section should be used for character classification.

21010 RATIONALE

21011 None.

21012 FUTURE DIRECTIONS

21013 None.

21014 SEE ALSO

21015 *isalnum()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
21016 *isxdigit()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
21017 <ctype.h>, <stdio.h>

21018 CHANGE HISTORY

21019 First released in Issue 1. Derived from Issue 1 of the SVID.

21020 Issue 6

21021 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

21022 NAME

21023 *isascii* — test for a 7-bit US-ASCII character

21024 SYNOPSIS

21025 XSI `#include <ctype.h>`
21026 `int isascii(int c);`
21027

21028 DESCRIPTION

21029 The *isascii()* function shall test whether *c* is a 7-bit US-ASCII character code.

21030 The *isascii()* function is defined on all integer values.

21031 RETURN VALUE

21032 The *isascii()* function shall return non-zero if *c* is a 7-bit US-ASCII character code between 0 and octal 0177 inclusive; otherwise, it shall return 0.

21034 ERRORS

21035 No errors are defined.

21036 EXAMPLES

21037 None.

21038 APPLICATION USAGE

21039 None.

21040 RATIONALE

21041 None.

21042 FUTURE DIRECTIONS

21043 None.

21044 SEE ALSO

21045 The Base Definitions volume of IEEE Std 1003.1-2001, *<ctype.h>*

21046 CHANGE HISTORY

21047 First released in Issue 1. Derived from Issue 1 of the SVID.

21048 NAME

21049 *isastream* — test a file descriptor (**STREAMS**)

21050 SYNOPSIS

21051 XSR

```
#include <stropts.h>
```

21052

```
int isastream(int fildes);
```

21053

21054 DESCRIPTION

21055 The *isastream()* function shall test whether *fildes*, an open file descriptor, is associated with a
21056 STREAMS-based file.

21057 RETURN VALUE

21058 Upon successful completion, *isastream()* shall return 1 if *fildes* refers to a STREAMS-based file
21059 and 0 if not. Otherwise, *isastream()* shall return -1 and set *errno* to indicate the error.

21060 ERRORS

21061 The *isastream()* function shall fail if:

21062 [EBADF] The *fildes* argument is not a valid open file descriptor.

21063 EXAMPLES

21064 None.

21065 APPLICATION USAGE

21066 None.

21067 RATIONALE

21068 None.

21069 FUTURE DIRECTIONS

21070 None.

21071 SEE ALSO

21072 The Base Definitions volume of IEEE Std 1003.1-2001, **<stropts.h>**

21073 CHANGE HISTORY

21074 First released in Issue 4, Version 2.

21075 Issue 5

21076 Moved from X/OPEN UNIX extension to BASE.

21077 NAME

21078 *isatty* — test for a terminal device

21079 SYNOPSIS

21080 #include <unistd.h>

21081 int isatty(int *fildes*);

21082 DESCRIPTION

21083 The *isatty*() function shall test whether *fildes*, an open file descriptor, is associated with a terminal device.

21085 RETURN VALUE

21086 The *isatty*() function shall return 1 if *fildes* is associated with a terminal; otherwise, it shall return 0 and may set *errno* to indicate the error.

21088 ERRORS

21089 The *isatty*() function may fail if:

21090 [EBADF] The *fildes* argument is not a valid open file descriptor.

21091 [ENOTTY] The *fildes* argument is not associated with a terminal.

21092 EXAMPLES

21093 None.

21094 APPLICATION USAGE

21095 The *isatty*() function does not necessarily indicate that a human being is available for interaction via *fildes*. It is quite possible that non-terminal devices are connected to the communications line.

21098 RATIONALE

21099 None.

21100 FUTURE DIRECTIONS

21101 None.

21102 SEE ALSO

21103 The Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

21104 CHANGE HISTORY

21105 First released in Issue 1. Derived from Issue 1 of the SVID.

21106 Issue 6

21107 The following new requirements on POSIX implementations derive from alignment with the
21108 Single UNIX Specification:

21109 • The optional setting of *errno* to indicate an error is added.

21110 • The [EBADF] and [ENOTTY] optional error conditions are added.

21111 NAME

21112 `isblank` — test for a blank character

21113 SYNOPSIS

```
21114        #include <ctype.h>
21115        int isblank(int c);
```

21116 DESCRIPTION

21117 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21118 conflict between the requirements described here and the ISO C standard is unintentional. This
21119 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21120 The `isblank()` function shall test whether *c* is a character of class **blank** in the program's current
21121 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21122 The *c* argument is a type **int**, the value of which the application shall ensure is a character
21123 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21124 any other value, the behavior is undefined.

21125 RETURN VALUE

21126 The `isblank()` function shall return non-zero if *c* is a <blank>; otherwise, it shall return 0.

21127 ERRORS

21128 No errors are defined.

21129 EXAMPLES

21130 None.

21131 APPLICATION USAGE

21132 To ensure applications portability, especially across natural languages, only this function and
21133 those listed in the SEE ALSO section should be used for character classification.

21134 RATIONALE

21135 None.

21136 FUTURE DIRECTIONS

21137 None.

21138 SEE ALSO

21139 `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`,
21140 `isxdigit()`, `setlocale()`, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
21141 <`ctype.h`>

21142 CHANGE HISTORY

21143 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21144 NAME

21145 *iscntrl* — test for a control character

21146 SYNOPSIS

```
21147        #include <ctype.h>
21148        int iscntrl(int c);
```

21149 DESCRIPTION

21150 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21151 conflict between the requirements described here and the ISO C standard is unintentional. This
21152 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21153 The *iscntrl()* function shall test whether *c* is a character of class **cntrl** in the program's current
21154 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21155 The *c* argument is a type **int**, the value of which the application shall ensure is a character
21156 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21157 any other value, the behavior is undefined.

21158 RETURN VALUE

21159 The *iscntrl()* function shall return non-zero if *c* is a control character; otherwise, it shall return 0.

21160 ERRORS

21161 No errors are defined.

21162 EXAMPLES

21163 None.

21164 APPLICATION USAGE

21165 To ensure applications portability, especially across natural languages, only this function and
21166 those listed in the SEE ALSO section should be used for character classification.

21167 RATIONALE

21168 None.

21169 FUTURE DIRECTIONS

21170 None.

21171 SEE ALSO

21172 *isalnum()*, *isalpha()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
21173 *isxdigit()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
21174 <ctype.h>

21175 CHANGE HISTORY

21176 First released in Issue 1. Derived from Issue 1 of the SVID.

21177 Issue 6

21178 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

21179 NAME

21180 *isdigit* — test for a decimal digit

21181 SYNOPSIS

```
21182     #include <ctype.h>
21183     int isdigit(int c);
```

21184 DESCRIPTION

21185 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21186 conflict between the requirements described here and the ISO C standard is unintentional. This
21187 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21188 The *isdigit()* function shall test whether *c* is a character of class **digit** in the program's current
21189 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21190 The *c* argument is an **int**, the value of which the application shall ensure is a character
21191 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21192 any other value, the behavior is undefined.

21193 RETURN VALUE

21194 The *isdigit()* function shall return non-zero if *c* is a decimal digit; otherwise, it shall return 0.

21195 ERRORS

21196 No errors are defined.

21197 EXAMPLES

21198 None.

21199 APPLICATION USAGE

21200 To ensure applications portability, especially across natural languages, only this function and
21201 those listed in the SEE ALSO section should be used for character classification.

21202 RATIONALE

21203 None.

21204 FUTURE DIRECTIONS

21205 None.

21206 SEE ALSO

21207 *isalnum()*, *isalpha()*, *iscntrl()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
21208 *isxdigit()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

21209 CHANGE HISTORY

21210 First released in Issue 1. Derived from Issue 1 of the SVID.

21211 Issue 6

21212 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

21213 NAME

21214 *isfinite* — test for finite value

21215 SYNOPSIS

```
21216        #include <math.h>
21217        int isfinite(real-floating x);
```

21218 DESCRIPTION

21219 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21220 conflict between the requirements described here and the ISO C standard is unintentional. This
21221 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21222 The *isfinite()* macro shall determine whether its argument has a finite value (zero, subnormal, or
21223 normal, and not infinite or NaN). First, an argument represented in a format wider than its
21224 semantic type is converted to its semantic type. Then determination is based on the type of the
21225 argument.

21226 RETURN VALUE

21227 The *isfinite()* macro shall return a non-zero value if and only if its argument has a finite value.

21228 ERRORS

21229 No errors are defined.

21230 EXAMPLES

21231 None.

21232 APPLICATION USAGE

21233 None.

21234 RATIONALE

21235 None.

21236 FUTURE DIRECTIONS

21237 None.

21238 SEE ALSO

21239 *fpclassify()*, *isinf()*, *isnan()*, *isnormal()*, *signbit()*, the Base Definitions volume of
21240 IEEE Std 1003.1-2001 <**math.h**>

21241 CHANGE HISTORY

21242 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21243 NAME

21244 `isgraph` — test for a visible character

21245 SYNOPSIS

```
21246        #include <ctype.h>
21247        int isgraph(int c);
```

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 `isgraph()` function shall test whether *c* is a character of class **graph** in the program's current
21253 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21254 The *c* argument is an **int**, the value of which the application shall ensure is a character
21255 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21256 any other value, the behavior is undefined.

21257 RETURN VALUE

21258 The `isgraph()` function shall return non-zero if *c* is a character with a visible representation;
21259 otherwise, it shall return 0.

21260 ERRORS

21261 No errors are defined.

21262 EXAMPLES

21263 None.

21264 APPLICATION USAGE

21265 To ensure applications portability, especially across natural languages, only this function and
21266 those listed in the SEE ALSO section should be used for character classification.

21267 RATIONALE

21268 None.

21269 FUTURE DIRECTIONS

21270 None.

21271 SEE ALSO

21272 `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`,
21273 `setlocale()`, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, `<ctype.h>`

21274 CHANGE HISTORY

21275 First released in Issue 1. Derived from Issue 1 of the SVID.

21276 Issue 6

21277 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

21278 NAME

21279 *isgreater* — test if *x* greater than *y*

21280 SYNOPSIS

```
21281        #include <math.h>
21282        int isgreater(real-floating x, real-floating y);
```

21283 DESCRIPTION

21284 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21285 conflict between the requirements described here and the ISO C standard is unintentional. This
21286 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21287 The *isgreater()* macro shall determine whether its first argument is greater than its second
21288 argument. The value of *isgreater(x, y)* shall be equal to $(x) > (y)$; however, unlike $(x) > (y)$,
21289 *isgreater(x, y)* shall not raise the invalid floating-point exception when *x* and *y* are unordered.

21290 RETURN VALUE

21291 Upon successful completion, the *isgreater()* macro shall return the value of $(x) > (y)$.

21292 If *x* or *y* is NaN, 0 shall be returned.

21293 ERRORS

21294 No errors are defined.

21295 EXAMPLES

21296 None.

21297 APPLICATION USAGE

21298 The relational and equality operators support the usual mathematical relationships between
21299 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21300 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21301 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21302 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21303 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21304 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21305 indicates that the argument shall be an expression of **real-floating** type.

21306 RATIONALE

21307 None.

21308 FUTURE DIRECTIONS

21309 None.

21310 SEE ALSO

21311 *isgreaterequal()*, *isless()*, *islessequal()*, *islessgreater()*, *isunordered()*, the Base Definitions volume of
21312 IEEE Std 1003.1-2001 <math.h>

21313 CHANGE HISTORY

21314 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21315 NAME

21316 *isgreaterequal* — test if *x* is greater than or equal to *y*

21317 SYNOPSIS

```
21318        #include <math.h>
21319        int isgreaterequal(real-floating x, real-floating y);
```

21320 DESCRIPTION

21321 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21322 conflict between the requirements described here and the ISO C standard is unintentional. This
21323 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21324 The *isgreaterequal()* macro shall determine whether its first argument is greater than or equal to
21325 its second argument. The value of *isgreaterequal(x, y)* shall be equal to $(x) \geq (y)$; however, unlike
21326 $(x) \geq (y)$, *isgreaterequal(x, y)* shall not raise the invalid floating-point exception when *x* and *y* are
21327 unordered.

21328 RETURN VALUE

21329 Upon successful completion, the *isgreaterequal()* macro shall return the value of $(x) \geq (y)$.

21330 If *x* or *y* is NaN, 0 shall be returned.

21331 ERRORS

21332 No errors are defined.

21333 EXAMPLES

21334 None.

21335 APPLICATION USAGE

21336 The relational and equality operators support the usual mathematical relationships between
21337 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21338 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21339 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21340 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21341 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21342 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21343 indicates that the argument shall be an expression of **real-floating** type.

21344 RATIONALE

21345 None.

21346 FUTURE DIRECTIONS

21347 None.

21348 SEE ALSO

21349 *isgreater()*, *isless()*, *islessequal()*, *islessgreater()*, *isunordered()*, the Base Definitions volume of
21350 IEEE Std 1003.1-2001 <math.h>

21351 CHANGE HISTORY

21352 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21353 NAME

21354 **isinf** — test for infinity

21355 SYNOPSIS

21356 #include <math.h>

21357 int isinf(real-floating x);

21358 DESCRIPTION

21359 CX 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.

21362 The *isinf()* macro shall determine whether its argument value is an infinity (positive or negative). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

21365 RETURN VALUE

21366 The *isinf()* macro shall return a non-zero value if and only if its argument has an infinite value.

21367 ERRORS

21368 No errors are defined.

21369 EXAMPLES

21370 None.

21371 APPLICATION USAGE

21372 None.

21373 RATIONALE

21374 None.

21375 FUTURE DIRECTIONS

21376 None.

21377 SEE ALSO

21378 *fpclassify()*, *isfinite()*, *isnan()*, *isnormal()*, *signbit()*, the Base Definitions volume of
21379 IEEE Std 1003.1-2001 <**math.h**>

21380 CHANGE HISTORY

21381 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21382 NAME

21383 **isless** — test if x is less than y

21384 SYNOPSIS

```
21385        #include <math.h>
21386        int isless(real-floating x, real-floating y);
```

21387 DESCRIPTION

21388 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21389 conflict between the requirements described here and the ISO C standard is unintentional. This
21390 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21391 The *isless()* macro shall determine whether its first argument is less than its second argument.
21392 The value of *isless(x, y)* shall be equal to $(x) < (y)$; however, unlike $(x) < (y)$, *isless(x, y)* shall not
21393 raise the invalid floating-point exception when *x* and *y* are unordered.

21394 RETURN VALUE

21395 Upon successful completion, the *isless()* macro shall return the value of $(x) < (y)$.

21396 If *x* or *y* is NaN, 0 shall be returned.

21397 ERRORS

21398 No errors are defined.

21399 EXAMPLES

21400 None.

21401 APPLICATION USAGE

21402 The relational and equality operators support the usual mathematical relationships between
21403 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21404 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21405 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21406 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21407 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21408 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21409 indicates that the argument shall be an expression of **real-floating** type.

21410 RATIONALE

21411 None.

21412 FUTURE DIRECTIONS

21413 None.

21414 SEE ALSO

21415 *isgreater()*, *isgreaterequal()*, *islessequal()*, *islessgreater()*, *isunordered()*, the Base Definitions volume
21416 of IEEE Std 1003.1-2001, <**math.h**>

21417 CHANGE HISTORY

21418 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21419 NAME

21420 *islessequal* — test if *x* is less than or equal to *y*

21421 SYNOPSIS

```
21422        #include <math.h>
21423        int islessequal(real-floating x, real-floating y);
```

21424 DESCRIPTION

21425 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21426 conflict between the requirements described here and the ISO C standard is unintentional. This
21427 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21428 The *islessequal()* macro shall determine whether its first argument is less than or equal to its
21429 second argument. The value of *islessequal(x, y)* shall be equal to $(x) \leq (y)$; however, unlike
21430 $(x) \leq (y)$, *islessequal(x, y)* shall not raise the invalid floating-point exception when *x* and *y* are
21431 unordered.

21432 RETURN VALUE

21433 Upon successful completion, the *islessequal()* macro shall return the value of $(x) \leq (y)$.

21434 If *x* or *y* is NaN, 0 shall be returned.

21435 ERRORS

21436 No errors are defined.

21437 EXAMPLES

21438 None.

21439 APPLICATION USAGE

21440 The relational and equality operators support the usual mathematical relationships between
21441 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21442 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21443 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21444 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21445 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21446 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21447 indicates that the argument shall be an expression of **real-floating** type.

21448 RATIONALE

21449 None.

21450 FUTURE DIRECTIONS

21451 None.

21452 SEE ALSO

21453 *isgreater()*, *isgreaterequal()*, *isless()*, *islessgreater()*, *isunordered()*, the Base Definitions volume of
21454 IEEE Std 1003.1-2001 <math.h>

21455 CHANGE HISTORY

21456 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21457 NAME

21458 *islessgreater* — test if *x* is less than or greater than *y*

21459 SYNOPSIS

21460 `#include <math.h>`

21461 `int islessgreater(real-floating x, real-floating y);`

21462 DESCRIPTION

21463 CX 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.

21466 The *islessgreater()* macro shall determine whether its first argument is less than or greater than its second argument. The *islessgreater(x, y)* macro is similar to $(x) < (y) \mid\mid (x) > (y)$; however, *islessgreater(x, y)* shall not raise the invalid floating-point exception when *x* and *y* are unordered (nor shall it evaluate *x* and *y* twice).

21470 RETURN VALUE

21471 Upon successful completion, the *islessgreater()* macro shall return the value of $(x) < (y) \mid\mid (x) > (y)$.

21473 If *x* or *y* is NaN, 0 shall be returned.

21474 ERRORS

21475 No errors are defined.

21476 EXAMPLES

21477 None.

21478 APPLICATION USAGE

21479 The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators may raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for NaNs without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating** indicates that the argument shall be an expression of **real-floating** type.

21487 RATIONALE

21488 None.

21489 FUTURE DIRECTIONS

21490 None.

21491 SEE ALSO

21492 *isgreater()*, *isgreaterequal()*, *isless()*, *islessequal()*, *isunordered()*, the Base Definitions volume of
21493 IEEE Std 1003.1-2001 **<math.h>**

21494 CHANGE HISTORY

21495 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21496 NAME

21497 islower — test for a lowercase letter

21498 SYNOPSIS

21499 #include <ctype.h>
21500 int islower(int c);

21501 DESCRIPTION

21502 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21503 conflict between the requirements described here and the ISO C standard is unintentional. This
21504 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.21505 The *islower()* function shall test whether *c* is a character of class **lower** in the program's current
21506 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.21507 The *c* argument is an **int**, the value of which the application shall ensure is a character
21508 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21509 any other value, the behavior is undefined.

21510 RETURN VALUE

21511 The *islower()* function shall return non-zero if *c* is a lowercase letter; otherwise, it shall return 0.

21512 ERRORS

21513 No errors are defined.

21514 EXAMPLES

21515 Testing for a Lowercase Letter

21516 The following example tests whether the value is a lowercase letter, based on the locale of the
21517 user, then uses it as part of a key value.21518 #include <ctype.h>
21519 #include <stdlib.h>
21520 #include <locale.h>
21521 ...
21522 char *keystr;
21523 int elementlen, len;
21524 char c;
21525 ...
21526 setlocale(LC_ALL, "");
21527 ...
21528 len = 0;
21529 while (len < elementlen) {
21530 c = (char) (rand() % 256);
21531 ...
21532 if (islower(c))
21533 keystr[len++] = c;
21534 }
21535 ...

21536 APPLICATION USAGE

21537 To ensure applications portability, especially across natural languages, only this function and
21538 those listed in the SEE ALSO section should be used for character classification.

21539 **RATIONALE**

21540 None.

21541 **FUTURE DIRECTIONS**

21542 None.

21543 **SEE ALSO**

21544 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *isprint()*, *ispunct()*, *isspace()*, *isupper()*,
21545 *isxdigit()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
21546 <**ctype.h**>

21547 **CHANGE HISTORY**

21548 First released in Issue 1. Derived from Issue 1 of the SVID.

21549 **Issue 6**

21550 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

21551 An example is added.

21552 NAME

21553 `isnan` — test for a NaN

21554 SYNOPSIS

```
21555        #include <math.h>  
21556        int isnan(real-floating x);
```

21557 DESCRIPTION

21558 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21559 conflict between the requirements described here and the ISO C standard is unintentional. This
21560 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21561 The `isnan()` macro shall determine whether its argument value is a NaN. First, an argument
21562 represented in a format wider than its semantic type is converted to its semantic type. Then
21563 determination is based on the type of the argument.

21564 RETURN VALUE

21565 The `isnan()` macro shall return a non-zero value if and only if its argument has a NaN value.

21566 ERRORS

21567 No errors are defined.

21568 EXAMPLES

21569 None.

21570 APPLICATION USAGE

21571 None.

21572 RATIONALE

21573 None.

21574 FUTURE DIRECTIONS

21575 None.

21576 SEE ALSO

21577 `fpclassify()`, `isfinite()`, `isinf()`, `isnormal()`, `signbit()`, the Base Definitions volume of
21578 IEEE Std 1003.1-2001, <`math.h`>

21579 CHANGE HISTORY

21580 First released in Issue 3.

21581 Issue 5

21582 The DESCRIPTION is updated to indicate the return value when NaN is not supported. This
21583 text was previously published in the APPLICATION USAGE section.

21584 Issue 6

21585 Re-written for alignment with the ISO/IEC 9899: 1999 standard.

21586 NAME

21587 *isnormal* — test for a normal value

21588 SYNOPSIS

```
21589        #include <math.h>
21590        int isnormal(real-floating x);
```

21591 DESCRIPTION

21592 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21593 conflict between the requirements described here and the ISO C standard is unintentional. This
21594 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21595 The *isnormal()* macro shall determine whether its argument value is normal (neither zero,
21596 subnormal, infinite, nor NaN). First, an argument represented in a format wider than its
21597 semantic type is converted to its semantic type. Then determination is based on the type of the
21598 argument.

21599 RETURN VALUE

21600 The *isnormal()* macro shall return a non-zero value if and only if its argument has a normal
21601 value.

21602 ERRORS

21603 No errors are defined.

21604 EXAMPLES

21605 None.

21606 APPLICATION USAGE

21607 None.

21608 RATIONALE

21609 None.

21610 FUTURE DIRECTIONS

21611 None.

21612 SEE ALSO

21613 *fpclassify()*, *isfinite()*, *isinf()*, *isnan()*, *signbit()*, the Base Definitions volume of
21614 IEEE Std 1003.1-2001, <**math.h**>

21615 CHANGE HISTORY

21616 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21617 NAME

21618 **isprint** — test for a printable character

21619 SYNOPSIS

```
21620        #include <ctype.h>
21621        int isprint(int c);
```

21622 DESCRIPTION

21623 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21624 conflict between the requirements described here and the ISO C standard is unintentional. This
21625 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21626 The *isprint()* function shall test whether *c* is a character of class **print** in the program's current
21627 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21628 The *c* argument is an **int**, the value of which the application shall ensure is a character
21629 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21630 any other value, the behavior is undefined.

21631 RETURN VALUE

21632 The *isprint()* function shall return non-zero if *c* is a printable character; otherwise, it shall return
21633 0.

21634 ERRORS

21635 No errors are defined.

21636 EXAMPLES

21637 None.

21638 APPLICATION USAGE

21639 To ensure applications portability, especially across natural languages, only this function and
21640 those listed in the SEE ALSO section should be used for character classification.

21641 RATIONALE

21642 None.

21643 FUTURE DIRECTIONS

21644 None.

21645 SEE ALSO

21646 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *ispunct()*, *isspace()*, *isupper()*,
21647 *isxdigit()*, *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
21648 <ctype.h>

21649 CHANGE HISTORY

21650 First released in Issue 1. Derived from Issue 1 of the SVID.

21651 Issue 6

21652 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

21653 NAME

21654 ispunct — test for a punctuation character

21655 SYNOPSIS

```
21656       #include <ctype.h>
21657       int ispunct(int c);
```

21658 DESCRIPTION

21659 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21660 conflict between the requirements described here and the ISO C standard is unintentional. This
21661 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21662 The *ispunct()* function shall test whether *c* is a character of class **punct** in the program's current
21663 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21664 The *c* argument is an **int**, the value of which the application shall ensure is a character
21665 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21666 any other value, the behavior is undefined.

21667 RETURN VALUE

21668 The *ispunct()* function shall return non-zero if *c* is a punctuation character; otherwise, it shall
21669 return 0.

21670 ERRORS

21671 No errors are defined.

21672 EXAMPLES

21673 None.

21674 APPLICATION USAGE

21675 To ensure applications portability, especially across natural languages, only this function and
21676 those listed in the SEE ALSO section should be used for character classification.

21677 RATIONALE

21678 None.

21679 FUTURE DIRECTIONS

21680 None.

21681 SEE ALSO

21682 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *isspace()*, *isupper()*, *isxdigit()*,
21683 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, **<ctype.h>**

21684 CHANGE HISTORY

21685 First released in Issue 1. Derived from Issue 1 of the SVID.

21686 Issue 6

21687 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

21688 NAME

21689 `isspace` — test for a white-space character

21690 SYNOPSIS

```
21691       #include <ctype.h>
21692       int isspace(int c);
```

21693 DESCRIPTION

21694 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21695 conflict between the requirements described here and the ISO C standard is unintentional. This
21696 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21697 The `isspace()` function shall test whether *c* is a character of class **space** in the program's current
21698 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21699 The *c* argument is an **int**, the value of which the application shall ensure is a character
21700 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21701 any other value, the behavior is undefined.

21702 RETURN VALUE

21703 The `isspace()` function shall return non-zero if *c* is a white-space character; otherwise, it shall
21704 return 0.

21705 ERRORS

21706 No errors are defined.

21707 EXAMPLES

21708 None.

21709 APPLICATION USAGE

21710 To ensure applications portability, especially across natural languages, only this function and
21711 those listed in the SEE ALSO section should be used for character classification.

21712 RATIONALE

21713 None.

21714 FUTURE DIRECTIONS

21715 None.

21716 SEE ALSO

21717 `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isupper()`,
21718 `isxdigit()`, `setlocale()`, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
21719 `<ctype.h>`

21720 CHANGE HISTORY

21721 First released in Issue 1. Derived from Issue 1 of the SVID.

21722 Issue 6

21723 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

21724 NAME

21725 *isunordered* — test if arguments are unordered

21726 SYNOPSIS

```
21727       #include <math.h>
21728       int isunordered(real-floating x, real-floating y);
```

21729 DESCRIPTION

21730 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21731 conflict between the requirements described here and the ISO C standard is unintentional. This
21732 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21733 The *isunordered()* macro shall determine whether its arguments are unordered.

21734 RETURN VALUE

21735 Upon successful completion, the *isunordered()* macro shall return 1 if its arguments are
21736 unordered, and 0 otherwise.

21737 If *x* or *y* is NaN, 1 shall be returned.

2

21738 ERRORS

21739 No errors are defined.

21740 EXAMPLES

21741 None.

21742 APPLICATION USAGE

21743 The relational and equality operators support the usual mathematical relationships between
21744 numeric values. For any ordered pair of numeric values, exactly one of the relationships (less,
21745 greater, and equal) is true. Relational operators may raise the invalid floating-point exception
21746 when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the
21747 unordered relationship is true. This macro is a quiet (non-floating-point exception raising)
21748 version of a relational operator. It facilitates writing efficient code that accounts for NaNs
21749 without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating**
21750 indicates that the argument shall be an expression of **real-floating** type.

21751 RATIONALE

21752 None.

21753 FUTURE DIRECTIONS

21754 None.

21755 SEE ALSO

21756 *isgreater()*, *isgreaterequal()*, *isless()*, *islessequal()*, *islessgreater()*, the Base Definitions volume of
21757 IEEE Std 1003.1-2001, <**math.h**>

21758 CHANGE HISTORY

21759 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21760 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/50 is applied, correcting the RETURN 2
21761 VALUE section when *x* or *y* is NaN. 2

21762 NAME

21763 *isupper* — test for an uppercase letter

21764 SYNOPSIS

```
21765        #include <ctype.h>
21766        int isupper(int c);
```

21767 DESCRIPTION

21768 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21769 conflict between the requirements described here and the ISO C standard is unintentional. This
21770 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21771 The *isupper()* function shall test whether *c* is a character of class **upper** in the program's current
21772 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

21773 The *c* argument is an **int**, the value of which the application shall ensure is a character
21774 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
21775 any other value, the behavior is undefined.

21776 RETURN VALUE

21777 The *isupper()* function shall return non-zero if *c* is an uppercase letter; otherwise, it shall return 0.

21778 ERRORS

21779 No errors are defined.

21780 EXAMPLES

21781 None.

21782 APPLICATION USAGE

21783 To ensure applications portability, especially across natural languages, only this function and
21784 those listed in the SEE ALSO section should be used for character classification.

21785 RATIONALE

21786 None.

21787 FUTURE DIRECTIONS

21788 None.

21789 SEE ALSO

21790 *isalnum()*, *isalpha()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*, *isspace()*, *isxdigit()*,
21791 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, **<ctype.h>**

21792 CHANGE HISTORY

21793 First released in Issue 1. Derived from Issue 1 of the SVID.

21794 Issue 6

21795 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

21796 NAME

21797 *iswalnum* — test for an alphanumeric wide-character code

21798 SYNOPSIS

```
21799        #include <wctype.h>
21800        int iswalnum(wint_t wc);
```

21801 DESCRIPTION

21802 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21803 conflict between the requirements described here and the ISO C standard is unintentional. This
21804 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21805 The *iswalnum()* function shall test whether *wc* is a wide-character code representing a character
21806 of class **alpha** or **digit** in the program's current locale; see the Base Definitions volume of
21807 IEEE Std 1003.1-2001, Chapter 7, Locale.

21808 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
21809 code corresponding to a valid character in the current locale, or equal to the value of the macro
21810 WEOF. If the argument has any other value, the behavior is undefined.

21811 RETURN VALUE

21812 The *iswalnum()* function shall return non-zero if *wc* is an alphanumeric wide-character code;
21813 otherwise, it shall return 0.

21814 ERRORS

21815 No errors are defined.

21816 EXAMPLES

21817 None.

21818 APPLICATION USAGE

21819 To ensure applications portability, especially across natural languages, only this function and
21820 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21821 RATIONALE

21822 None.

21823 FUTURE DIRECTIONS

21824 None.

21825 SEE ALSO

21826 *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*,
21827 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
21828 IEEE Std 1003.1-2001, Chapter 7, Locale, <stdio.h>, <wchar.h>, <wctype.h>

21829 CHANGE HISTORY

21830 First released as a World-wide Portability Interface in Issue 4.

21831 Issue 5

21832 The following change has been made in this issue for alignment with
21833 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21834 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21835 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

21836 Issue 6

21837 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

21838 NAME

21839 *iswalph* — test for an alphabetic wide-character code

21840 SYNOPSIS

```
21841        #include <wctype.h>
21842        int iswalph(wint_t wc);
```

21843 DESCRIPTION

21844 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21845 conflict between the requirements described here and the ISO C standard is unintentional. This
21846 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21847 The *iswalph*() function shall test whether *wc* is a wide-character code representing a character of
21848 class **alpha** in the program's current locale; see the Base Definitions volume of
21849 IEEE Std 1003.1-2001, Chapter 7, Locale.

21850 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
21851 code corresponding to a valid character in the current locale, or equal to the value of the macro
21852 WEOF. If the argument has any other value, the behavior is undefined.

21853 RETURN VALUE

21854 The *iswalph*() function shall return non-zero if *wc* is an alphabetic wide-character code;
21855 otherwise, it shall return 0.

21856 ERRORS

21857 No errors are defined.

21858 EXAMPLES

21859 None.

21860 APPLICATION USAGE

21861 To ensure applications portability, especially across natural languages, only this function and
21862 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21863 RATIONALE

21864 None.

21865 FUTURE DIRECTIONS

21866 None.

21867 SEE ALSO

21868 *iswalnum*(), *iswcntrl*(), *iswctype*(), *iswdigit*(), *iswgraph*(), *iswlower*(), *iswprint*(), *iswpunct*(),
21869 *iswspace*(), *iswupper*(), *iswxdigit*(), *setlocale*(), the Base Definitions volume of
21870 IEEE Std 1003.1-2001, Chapter 7, Locale, <stdio.h>, <wchar.h>, <wctype.h>

21871 CHANGE HISTORY

21872 First released in Issue 4.

21873 Issue 5

21874 The following change has been made in this issue for alignment with
21875 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21876 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21877 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

21878 Issue 6

21879 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

21880 NAME

21881 *iswblank* — test for a blank wide-character code

21882 SYNOPSIS

```
21883 #include <wctype.h>
21884 int iswblank(wint_t wc);
```

21885 DESCRIPTION

21886 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21887 conflict between the requirements described here and the ISO C standard is unintentional. This
21888 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21889 The *iswblank()* function shall test whether *wc* is a wide-character code representing a character of
21890 class **blank** in the program's current locale; see the Base Definitions volume of
21891 IEEE Std 1003.1-2001, Chapter 7, Locale.

21892 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
21893 code corresponding to a valid character in the current locale, or equal to the value of the macro
21894 WEOF. If the argument has any other value, the behavior is undefined.

21895 RETURN VALUE

21896 The *iswblank()* function shall return non-zero if *wc* is a blank wide-character code; otherwise, it
21897 shall return 0.

21898 ERRORS

21899 No errors are defined.

21900 EXAMPLES

21901 None.

21902 APPLICATION USAGE

21903 To ensure applications portability, especially across natural languages, only this function and
21904 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21905 RATIONALE

21906 None.

21907 FUTURE DIRECTIONS

21908 None.

21909 SEE ALSO

21910 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*,
21911 *iswpunct()*, *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
21912 IEEE Std 1003.1-2001, Chapter 7, Locale, **<stdio.h>**, **<wchar.h>**, **<wctype.h>**

21913 CHANGE HISTORY

21914 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

21915 NAME

21916 *iswcntrl* — test for a control wide-character code

21917 SYNOPSIS

```
21918        #include <wctype.h>
21919        int iswcntrl(wint_t wc);
```

21920 DESCRIPTION

21921 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21922 conflict between the requirements described here and the ISO C standard is unintentional. This
21923 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21924 The *iswcntrl()* function shall test whether *wc* is a wide-character code representing a character of
21925 class *cntrl* in the program's current locale; see the Base Definitions volume of
21926 IEEE Std 1003.1-2001, Chapter 7, Locale.

21927 The *wc* argument is a *wint_t*, the value of which the application shall ensure is a wide-character
21928 code corresponding to a valid character in the current locale, or equal to the value of the macro
21929 WEOF. If the argument has any other value, the behavior is undefined.

21930 RETURN VALUE

21931 The *iswcntrl()* function shall return non-zero if *wc* is a control wide-character code; otherwise, it
21932 shall return 0.

21933 ERRORS

21934 No errors are defined.

21935 EXAMPLES

21936 None.

21937 APPLICATION USAGE

21938 To ensure applications portability, especially across natural languages, only this function and
21939 those listed in the SEE ALSO section should be used for classification of wide-character codes.

21940 RATIONALE

21941 None.

21942 FUTURE DIRECTIONS

21943 None.

21944 SEE ALSO

21945 *iswalnum()*, *iswalpha()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*,
21946 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
21947 IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

21948 CHANGE HISTORY

21949 First released in Issue 4.

21950 Issue 5

21951 The following change has been made in this issue for alignment with
21952 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 21953 • The SYNOPSIS has been changed to indicate that this function and associated data types are
21954 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

21955 Issue 6

21956 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

21957 **NAME**

21958 `iswctype` — test character for a specified class

21959 **SYNOPSIS**

```
21960        #include <wctype.h>
21961        int iswctype(wint_t wc, wctype_t charclass);
```

21962 **DESCRIPTION**

21963 CX The functionality described on this reference page is aligned with the ISO C standard. Any
21964 conflict between the requirements described here and the ISO C standard is unintentional. This
21965 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

21966 The `iswctype()` function shall determine whether the wide-character code `wc` has the character
21967 class `charclass`, returning true or false. The `iswctype()` function is defined on WEOF and wide-
21968 character codes corresponding to the valid character encodings in the current locale. If the `wc`
21969 argument is not in the domain of the function, the result is undefined. If the value of `charclass` is
21970 invalid (that is, not obtained by a call to `wctype()` or `charclass` is invalidated by a subsequent call
21971 to `setlocale()` that has affected category `LC_CTYPE`) the result is unspecified.

21972 **RETURN VALUE**

21973 The `iswctype()` function shall return non-zero (true) if and only if `wc` has the property described
21974 CX by `charclass`. If `charclass` is 0, `iswctype()` shall return 0.

21975 **ERRORS**

21976 No errors are defined.

21977 **EXAMPLES**21978 **Testing for a Valid Character**

```
21979        #include <wctype.h>
21980        ...
21981        int yes_or_no;
21982        wint_t wc;
21983        wctype_t valid_class;
21984        ...
21985        if ((valid_class=wctype("vowel")) == (wctype_t)0)
21986          /* Invalid character class. */
21987        yes_or_no=iswctype(wc,valid_class);
```

21988 **APPLICATION USAGE**

21989 The twelve strings "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower",
21990 "print", "punct", "space", "upper", and "xdigit" are reserved for the standard
21991 character classes. In the table below, the functions in the left column are equivalent to the
21992 functions in the right column.

21993 <code>iswalnum(wc)</code>	21993 <code>iswctype(wc, wctype("alnum"))</code>
21994 <code>iswalpha(wc)</code>	21994 <code>iswctype(wc, wctype("alpha"))</code>
21995 <code>iswblank(wc)</code>	21995 <code>iswctype(wc, wctype("blank"))</code>
21996 <code>iswcntrl(wc)</code>	21996 <code>iswctype(wc, wctype("cntrl"))</code>
21997 <code>iswdigit(wc)</code>	21997 <code>iswctype(wc, wctype("digit"))</code>
21998 <code>iswgraph(wc)</code>	21998 <code>iswctype(wc, wctype("graph"))</code>
21999 <code>iswlower(wc)</code>	21999 <code>iswctype(wc, wctype("lower"))</code>
22000 <code>iswprint(wc)</code>	22000 <code>iswctype(wc, wctype("print"))</code>
22001 <code>iswpunct(wc)</code>	22001 <code>iswctype(wc, wctype("punct"))</code>
22002 <code>iswspace(wc)</code>	22002 <code>iswctype(wc, wctype("space"))</code>

22003 *iswupper(wc)* *iswctype(wc, wctype("upper"))*
22004 *iswxdigit(wc)* *iswctype(wc, wctype("xdigit"))*

22005 RATIONALE

22006 None.

22007 FUTURE DIRECTIONS

22008 None.

22009 SEE ALSO

22010 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*,
22011 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, *wctype()*, the Base Definitions volume of
22012 IEEE Std 1003.1-2001, <wchar.h>, <wctype.h>

22013 CHANGE HISTORY

22014 First released as World-wide Portability Interfaces in Issue 4.

22015 Issue 5

22016 The following change has been made in this issue for alignment with
22017 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 22018 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22019 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

22020 Issue 6

22021 The behavior of *n*=0 is now described.

22022 An example is added.

22023 A new function, *iswblank()*, is added to the list in the APPLICATION USAGE.

22024 NAME

22025 *iswdigit* — test for a decimal digit wide-character code

22026 SYNOPSIS

```
22027        #include <wctype.h>
22028        int iswdigit(wint_t wc);
```

22029 DESCRIPTION

22030 CX 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.

22033 The *iswdigit()* function shall test whether *wc* is a wide-character code representing a character of class **digit** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

22036 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro **WEOF**. If the argument has any other value, the behavior is undefined.

22039 RETURN VALUE

22040 The *iswdigit()* function shall return non-zero if *wc* is a decimal digit wide-character code; 22041 otherwise, it shall return 0.

22042 ERRORS

22043 No errors are defined.

22044 EXAMPLES

22045 None.

22046 APPLICATION USAGE

22047 To ensure applications portability, especially across natural languages, only this function and 22048 those listed in the SEE ALSO section should be used for classification of wide-character codes.

22049 RATIONALE

22050 None.

22051 FUTURE DIRECTIONS

22052 None.

22053 SEE ALSO

22054 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*,
22055 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
22056 IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

22057 CHANGE HISTORY

22058 First released in Issue 4.

22059 Issue 5

22060 The following change has been made in this issue for alignment with
22061 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 22062 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22063 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

22064 Issue 6

22065 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

22066 NAME

22067 *iswgraph* — test for a visible wide-character code

22068 SYNOPSIS

```
22069     #include <wctype.h>
22070     int iswgraph(wint_t wc);
```

22071 DESCRIPTION

22072 CX 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.

22075 The *iswgraph()* function shall test whether *wc* is a wide-character code representing a character of class **graph** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

22078 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro **WEOF**. If the argument has any other value, the behavior is undefined.

22081 RETURN VALUE

22082 The *iswgraph()* function shall return non-zero if *wc* is a wide-character code with a visible representation; otherwise, it shall return 0.

22084 ERRORS

22085 No errors are defined.

22086 EXAMPLES

22087 None.

22088 APPLICATION USAGE

22089 To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

22091 RATIONALE

22092 None.

22093 FUTURE DIRECTIONS

22094 None.

22095 SEE ALSO

22096 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswlower()*, *iswprint()*, *iswpunct()*,
22097 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
22098 IEEE Std 1003.1-2001, Chapter 7, Locale, <**wchar.h**>, <**wctype.h**>

22099 CHANGE HISTORY

22100 First released in Issue 4.

22101 Issue 5

22102 The following change has been made in this issue for alignment with
22103 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 22104 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22105 now made visible by inclusion of the <**wctype.h**> header rather than <**wchar.h**>.

22106 Issue 6

22107 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

22108 NAME

22109 *iswlower* — test for a lowercase letter wide-character code

22110 SYNOPSIS

```
22111        #include <wctype.h>
22112        int iswlower(wint_t wc);
```

22113 DESCRIPTION

22114 CX 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.

22117 The *iswlower()* function shall test whether *wc* is a wide-character code representing a character of class **lower** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

22120 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro **WEOF**. If the argument has any other value, the behavior is undefined.

22123 RETURN VALUE

22124 The *iswlower()* function shall return non-zero if *wc* is a lowercase letter wide-character code; otherwise, it shall return 0.

22126 ERRORS

22127 No errors are defined.

22128 EXAMPLES

22129 None.

22130 APPLICATION USAGE

22131 To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

22133 RATIONALE

22134 None.

22135 FUTURE DIRECTIONS

22136 None.

22137 SEE ALSO

22138 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswprint()*, *iswpunct()*,
22139 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
22140 IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

22141 CHANGE HISTORY

22142 First released in Issue 4.

22143 Issue 5

22144 The following change has been made in this issue for alignment with
22145 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 22146 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22147 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

22148 Issue 6

22149 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

22150 NAME

22151 *iswprint* — test for a printable wide-character code

22152 SYNOPSIS

```
22153        #include <wctype.h>
22154        int iswprint(wint_t wc);
```

22155 DESCRIPTION

22156 CX 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.

22159 The *iswprint()* function shall test whether *wc* is a wide-character code representing a character of class **print** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

22162 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro **WEOF**. If the argument has any other value, the behavior is undefined.

22165 RETURN VALUE

22166 The *iswprint()* function shall return non-zero if *wc* is a printable wide-character code; otherwise, it shall return 0.

22168 ERRORS

22169 No errors are defined.

22170 EXAMPLES

22171 None.

22172 APPLICATION USAGE

22173 To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

22175 RATIONALE

22176 None.

22177 FUTURE DIRECTIONS

22178 None.

22179 SEE ALSO

22180 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswpunct()*,
22181 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
22182 IEEE Std 1003.1-2001, Chapter 7, Locale, <**wchar.h**>, <**wctype.h**>

22183 CHANGE HISTORY

22184 First released in Issue 4.

22185 Issue 5

22186 The following change has been made in this issue for alignment with
22187 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 22188 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22189 now made visible by inclusion of the <**wctype.h**> header rather than <**wchar.h**>.

22190 Issue 6

22191 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

22192 NAME

22193 *iswpunct* — test for a punctuation wide-character code

22194 SYNOPSIS

```
22195        #include <wctype.h>
22196        int iswpunct(wint_t wc);
```

22197 DESCRIPTION

22198 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22199 conflict between the requirements described here and the ISO C standard is unintentional. This
22200 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

22201 The *iswpunct()* function shall test whether *wc* is a wide-character code representing a character
22202 of class **punct** in the program's current locale; see the Base Definitions volume of
22203 IEEE Std 1003.1-2001, Chapter 7, Locale.

22204 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22205 code corresponding to a valid character in the current locale, or equal to the value of the macro
22206 WEOF. If the argument has any other value, the behavior is undefined.

22207 RETURN VALUE

22208 The *iswpunct()* function shall return non-zero if *wc* is a punctuation wide-character code;
22209 otherwise, it shall return 0.

22210 ERRORS

22211 No errors are defined.

22212 EXAMPLES

22213 None.

22214 APPLICATION USAGE

22215 To ensure applications portability, especially across natural languages, only this function and
22216 those listed in the SEE ALSO section should be used for classification of wide-character codes.

22217 RATIONALE

22218 None.

22219 FUTURE DIRECTIONS

22220 None.

22221 SEE ALSO

22222 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*,
22223 *iswspace()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
22224 IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

22225 CHANGE HISTORY

22226 First released in Issue 4.

22227 Issue 5

22228 The following change has been made in this issue for alignment with
22229 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 22230 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22231 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

22232 Issue 6

22233 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

22234 NAME

22235 *iswspace* — test for a white-space wide-character code

22236 SYNOPSIS

```
22237        #include <wctype.h>
22238        int iswspace(wint_t wc);
```

22239 DESCRIPTION

22240 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22241 conflict between the requirements described here and the ISO C standard is unintentional. This
22242 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

22243 The *iswspace()* function shall test whether *wc* is a wide-character code representing a character of
22244 class **space** in the program's current locale; see the Base Definitions volume of
22245 IEEE Std 1003.1-2001, Chapter 7, Locale.

22246 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character
22247 code corresponding to a valid character in the current locale, or equal to the value of the macro
22248 **WEOF**. If the argument has any other value, the behavior is undefined.

22249 RETURN VALUE

22250 The *iswspace()* function shall return non-zero if *wc* is a white-space wide-character code;
22251 otherwise, it shall return 0.

22252 ERRORS

22253 No errors are defined.

22254 EXAMPLES

22255 None.

22256 APPLICATION USAGE

22257 To ensure applications portability, especially across natural languages, only this function and
22258 those listed in the SEE ALSO section should be used for classification of wide-character codes.

22259 RATIONALE

22260 None.

22261 FUTURE DIRECTIONS

22262 None.

22263 SEE ALSO

22264 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*,
22265 *iswpunct()*, *iswupper()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
22266 IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

22267 CHANGE HISTORY

22268 First released in Issue 4.

22269 Issue 5

22270 The following change has been made in this issue for alignment with
22271 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 22272 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22273 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

22274 Issue 6

22275 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

22276 NAME

22277 *iswupper* — test for an uppercase letter wide-character code

22278 SYNOPSIS

```
22279        #include <wctype.h>
22280        int iswupper(wint_t wc);
```

22281 DESCRIPTION

22282 CX 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.

22285 The *iswupper()* function shall test whether *wc* is a wide-character code representing a character of class **upper** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

22288 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro **WEOF**. If the argument has any other value, the behavior is undefined.

22291 RETURN VALUE

22292 The *iswupper()* function shall return non-zero if *wc* is an uppercase letter wide-character code; otherwise, it shall return 0.

22294 ERRORS

22295 No errors are defined.

22296 EXAMPLES

22297 None.

22298 APPLICATION USAGE

22299 To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

22301 RATIONALE

22302 None.

22303 FUTURE DIRECTIONS

22304 None.

22305 SEE ALSO

22306 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*,
22307 *iswpunct()*, *iswspace()*, *iswxdigit()*, *setlocale()*, the Base Definitions volume of
22308 IEEE Std 1003.1-2001, Chapter 7, Locale, <**wchar.h**>, <**wctype.h**>

22309 CHANGE HISTORY

22310 First released in Issue 4.

22311 Issue 5

22312 The following change has been made in this issue for alignment with
22313 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 22314 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22315 now made visible by inclusion of the <**wctype.h**> header rather than <**wchar.h**>.

22316 Issue 6

22317 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

22318 NAME

22319 *iswxdigit* — test for a hexadecimal digit wide-character code

22320 SYNOPSIS

```
22321        #include <wctype.h>
22322        int iswxdigit(wint_t wc);
```

22323 DESCRIPTION

22324 CX 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.

22327 The *iswxdigit()* function shall test whether *wc* is a wide-character code representing a character of class **xdigit** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

22330 The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro **WEOF**. If the argument has any other value, the behavior is undefined.

22333 RETURN VALUE

22334 The *iswxdigit()* function shall return non-zero if *wc* is a hexadecimal digit wide-character code; otherwise, it shall return 0.

22336 ERRORS

22337 No errors are defined.

22338 EXAMPLES

22339 None.

22340 APPLICATION USAGE

22341 To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

22343 RATIONALE

22344 None.

22345 FUTURE DIRECTIONS

22346 None.

22347 SEE ALSO

22348 *iswalnum()*, *iswalpha()*, *iswcntrl()*, *iswctype()*, *iswdigit()*, *iswgraph()*, *iswlower()*, *iswprint()*,
22349 *iswpunct()*, *iswspace()*, *iswupper()*, *setlocale()*, the Base Definitions volume of
22350 IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

22351 CHANGE HISTORY

22352 First released in Issue 4.

22353 Issue 5

22354 The following change has been made in this issue for alignment with
22355 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 22356 • The SYNOPSIS has been changed to indicate that this function and associated data types are
22357 now made visible by inclusion of the <wctype.h> header rather than <wchar.h>.

22358 Issue 6

22359 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

22360 NAME

22361 `isxdigit` — test for a hexadecimal digit

22362 SYNOPSIS

```
22363        #include <ctype.h>
22364        int isxdigit(int c);
```

22365 DESCRIPTION

22366 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22367 conflict between the requirements described here and the ISO C standard is unintentional. This
22368 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

22369 The `isxdigit()` function shall test whether *c* is a character of class **xdigit** in the program's current
22370 locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

22371 The *c* argument is an **int**, the value of which the application shall ensure is a character
22372 representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has
22373 any other value, the behavior is undefined.

22374 RETURN VALUE

22375 The `isxdigit()` function shall return non-zero if *c* is a hexadecimal digit; otherwise, it shall return
22376 0.

22377 ERRORS

22378 No errors are defined.

22379 EXAMPLES

22380 None.

22381 APPLICATION USAGE

22382 To ensure applications portability, especially across natural languages, only this function and
22383 those listed in the SEE ALSO section should be used for character classification.

22384 RATIONALE

22385 None.

22386 FUTURE DIRECTIONS

22387 None.

22388 SEE ALSO

22389 `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`,
22390 the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, `<ctype.h>`

22391 CHANGE HISTORY

22392 First released in Issue 1. Derived from Issue 1 of the SVID.

22393 Issue 6

22394 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

22395 NAME

22396 j0, j1, jn — Bessel functions of the first kind

22397 SYNOPSIS

22398 XSI #include <math.h>

```
22399 double j0(double x);  
22400 double j1(double x);  
22401 double jn(int n, double x);  
22402
```

22403 DESCRIPTION

22404 The *j0()*, *j1()*, and *jn()* functions shall compute Bessel functions of *x* of the first kind of orders 0, 1, and *n*, respectively.

22406 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.

22410 RETURN VALUE

22411 Upon successful completion, these functions shall return the relevant Bessel value of *x* of the
22412 first kind.

22413 If the *x* argument is too large in magnitude, or the correct result would cause underflow, 0 shall
22414 be returned and a range error may occur.

22415 If *x* is NaN, a NaN shall be returned.

22416 ERRORS

22417 These functions may fail if:

22418 Range Error The value of *x* was too large in magnitude, or an underflow occurred.

22419 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
22420 then *errno* shall be set to [ERANGE]. If the integer expression
22421 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
22422 floating-point exception shall be raised.

22423 No other errors shall occur.

22424 EXAMPLES

22425 None.

22426 APPLICATION USAGE

22427 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
22428 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

22429 RATIONALE

22430 None.

22431 FUTURE DIRECTIONS

22432 None.

22433 SEE ALSO

22434 *feclearexcept()*, *fetestexcept()*, *isnan()*, *y0()*, the Base Definitions volume of IEEE Std 1003.1-2001,
22435 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

22436 CHANGE HISTORY

22437 First released in Issue 1. Derived from Issue 1 of the SVID.

22438 Issue 5

22439 The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

22441 Issue 6

22442 The may fail [EDOM] error is removed for the case for NaN.

22443 The RETURN VALUE and ERRORS sections are reworked for alignment of the error handling
22444 with the ISO/IEC 9899:1999 standard.

22445 NAME

22446 jrand48 — generate a uniformly distributed pseudo-random long signed integer

22447 SYNOPSIS

22448 XSI #include <stdlib.h>

22449 long jrand48(unsigned short xsubi[3]);

22450

22451 DESCRIPTION

22452 Refer to *drand48()*.

22453 NAME

22454 kill — send a signal to a process or a group of processes

22455 SYNOPSIS

22456 CX #include <signal.h>

22457 int kill(pid_t pid, int sig);

22458

22459 DESCRIPTION

22460 The *kill()* function shall send a signal to a process or a group of processes specified by *pid*. The
22461 signal to be sent is specified by *sig* and is either one from the list given in <signal.h> or 0. If *sig* is
22462 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can
22463 be used to check the validity of *pid*.

22464 For a process to have permission to send a signal to a process designated by *pid*, unless the
22465 sending process has appropriate privileges, the real or effective user ID of the sending process
22466 shall match the real or saved set-user-ID of the receiving process.

22467 If *pid* is greater than 0, *sig* shall be sent to the process whose process ID is equal to *pid*.

22468 If *pid* is 0, *sig* shall be sent to all processes (excluding an unspecified set of system processes)
22469 whose process group ID is equal to the process group ID of the sender, and for which the
22470 process has permission to send a signal.

22471 If *pid* is -1, *sig* shall be sent to all processes (excluding an unspecified set of system processes) for
22472 which the process has permission to send that signal.

22473 If *pid* is negative, but not -1, *sig* shall be sent to all processes (excluding an unspecified set of
22474 system processes) whose process group ID is equal to the absolute value of *pid*, and for which
22475 the process has permission to send a signal.

22476 If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked for
22477 the calling thread and if no other thread has *sig* unblocked or is waiting in a *sigwait()* function
22478 for *sig*, either *sig* or at least one pending unblocked signal shall be delivered to the sending
22479 thread before *kill()* returns.

22480 The user ID tests described above shall not be applied when sending SIGCONT to a process that
22481 is a member of the same session as the sending process.

22482 An implementation that provides extended security controls may impose further
22483 implementation-defined restrictions on the sending of signals, including the null signal. In
22484 particular, the system may deny the existence of some or all of the processes specified by *pid*.

22485 The *kill()* function is successful if the process has permission to send *sig* to any of the processes
22486 specified by *pid*. If *kill()* fails, no signal shall be sent.

22487 RETURN VALUE

22488 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
22489 indicate the error.

22490 ERRORS

22491 The *kill()* function shall fail if:

22492 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number.

22493 [EPERM] The process does not have permission to send the signal to any receiving
22494 process.

22495 [ESRCH] No process or process group can be found corresponding to that specified by
22496 *pid*.

22497 EXAMPLES

22498 None.

22499 APPLICATION USAGE

22500 None.

22501 RATIONALE

22502 The semantics for permission checking for *kill()* differed between System V and most other
22503 implementations, such as Version 7 or 4.3 BSD. The semantics chosen for this volume of
22504 IEEE Std 1003.1-2001 agree with System V. Specifically, a set-user-ID process cannot protect
22505 itself against signals (or at least not against SIGKILL) unless it changes its real user ID. This
22506 choice allows the user who starts an application to send it signals even if it changes its effective
22507 user ID. The other semantics give more power to an application that wants to protect itself from
22508 the user who ran it.

22509 Some implementations provide semantic extensions to the *kill()* function when the absolute
22510 value of *pid* is greater than some maximum, or otherwise special, value. Negative values are a
22511 flag to *kill()*. Since most implementations return [ESRCH] in this case, this behavior is not
22512 included in this volume of IEEE Std 1003.1-2001, although a conforming implementation could
22513 provide such an extension.

22514 The unspecified processes to which a signal cannot be sent may include the scheduler or *init*.

2

22515 There was initially strong sentiment to specify that, if *pid* specifies that a signal be sent to the
22516 calling process and that signal is not blocked, that signal would be delivered before *kill()*
22517 returns. This would permit a process to call *kill()* and be guaranteed that the call never return.
22518 However, historical implementations that provide only the *signal()* function make only the
22519 weaker guarantee in this volume of IEEE Std 1003.1-2001, because they only deliver one signal
22520 each time a process enters the kernel. Modifications to such implementations to support the
22521 *sigaction()* function generally require entry to the kernel following return from a signal-catching
22522 function, in order to restore the signal mask. Such modifications have the effect of satisfying the
22523 stronger requirement, at least when *sigaction()* is used, but not necessarily when *signal()* is used.
22524 The developers of this volume of IEEE Std 1003.1-2001 considered making the stronger
22525 requirement except when *signal()* is used, but felt this would be unnecessarily complex.
22526 Implementors are encouraged to meet the stronger requirement whenever possible. In practice,
22527 the weaker requirement is the same, except in the rare case when two signals arrive during a
22528 very short window. This reasoning also applies to a similar requirement for *sigprocmask()*.

22529 In 4.2 BSD, the SIGCONT signal can be sent to any descendant process regardless of user-ID
22530 security checks. This allows a job control shell to continue a job even if processes in the job have
22531 altered their user IDs (as in the *su* command). In keeping with the addition of the concept of
22532 sessions, similar functionality is provided by allowing the SIGCONT signal to be sent to any
22533 process in the same session regardless of user ID security checks. This is less restrictive than BSD
22534 in the sense that ancestor processes (in the same session) can now be the recipient. It is more
22535 restrictive than BSD in the sense that descendant processes that form new sessions are now
22536 subject to the user ID checks. A similar relaxation of security is not necessary for the other job
22537 control signals since those signals are typically sent by the terminal driver in recognition of
22538 special characters being typed; the terminal driver bypasses all security checks.

22539 In secure implementations, a process may be restricted from sending a signal to a process having
22540 a different security label. In order to prevent the existence or nonexistence of a process from
22541 being used as a covert channel, such processes should appear nonexistent to the sender; that is,
22542 [ESRCH] should be returned, rather than [EPERM], if *pid* refers only to such processes.

22543 Existing implementations vary on the result of a *kill()* with *pid* indicating an inactive process (a
22544 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.

22554 FUTURE DIRECTIONS

22555 None.

22556 SEE ALSO

22557 *getpid()*, *raise()*, *setsid()*, *sigaction()*, *sigqueue()*, the Base Definitions volume of
22558 IEEE Std 1003.1-2001, <**signal.h**>, <**sys/types.h**>

22559 CHANGE HISTORY

22560 First released in Issue 1. Derived from Issue 1 of the SVID.

22561 Issue 5

22562 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

22563 Issue 6

22564 In the SYNOPSIS, the optional include of the <**sys/types.h**> header is removed.

22565 The following new requirements on POSIX implementations derive from alignment with the
22566 Single UNIX Specification:

- 22567 • In the DESCRIPTION, the second paragraph is reworded to indicate that the saved set-user-
22568 ID of the calling process is checked in place of its effective user ID. This is a FIPS
22569 requirement.
- 22570 • The requirement to include <**sys/types.h**> has been removed. Although <**sys/types.h**> was
22571 required for conforming implementations of previous POSIX specifications, it was not
22572 required for UNIX applications.
- 22573 • The behavior when *pid* is -1 is now specified. It was previously explicitly unspecified in the
22574 POSIX.1-1988 standard.

22575 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

22576 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/51 is applied, correcting the RATIONALE 2
22577 section. 2

22578 **NAME**

22579 killpg — send a signal to a process group

22580 **SYNOPSIS**

22581 XSI #include <signal.h>

22582 int killpg(pid_t pgrp, int sig);

22583

22584 **DESCRIPTION**

22585 The *killpg()* function shall send the signal specified by *sig* to the process group specified by *pgrp*.

22586 If *pgrp* is greater than 1, *killpg(pgrp, sig)* shall be equivalent to *kill(-pgrp, sig)*. If *pgrp* is less than or
22587 equal to 1, the behavior of *killpg()* is undefined.

22588 **RETURN VALUE**

22589 Refer to *kill()*.

22590 **ERRORS**

22591 Refer to *kill()*.

22592 **EXAMPLES**22593 **Sending a Signal to All Other Members of a Process Group**

2

22594 The following example shows how the calling process could send a signal to all other members
22595 of its process group. To prevent itself from receiving the signal it first makes itself immune to the
22596 signal by ignoring it.

2

2

2

```
22597       #include <signal.h>
22598       #include <unistd.h>
22599       ...
22600       if (signal(SIGUSR1, SIG_IGN) == SIG_ERR)
22601           /* Handle error */;
22602       if (killpg(getpgrp(), SIGUSR1) == -1)
22603           /* Handle error */;"
```

2

2

2

22604 **APPLICATION USAGE**

22605 None.

22606 **RATIONALE**

22607 None.

22608 **FUTURE DIRECTIONS**

22609 None.

22610 **SEE ALSO**

22611 *getpgid()*, *getpid()*, *kill()*, *raise()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>

22612 **CHANGE HISTORY**

22613 First released in Issue 4, Version 2.

22614 **Issue 5**

22615 Moved from X/OPEN UNIX extension to BASE.

22616 **Issue 6**

2

22617 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/52 is applied, adding the example to the
22618 EXAMPLES section.

2

2

22619 **NAME**

22620 l64a — convert a 32-bit integer to a radix-64 ASCII string

22621 **SYNOPSIS**

22622 XSI #include <stdlib.h>

22623 char *l64a(long value);

22624

22625 **DESCRIPTION**

22626 Refer to *a64l()*.

22627 NAME

22628 **labs, llabs** — return a long integer absolute value

22629 SYNOPSIS

```
22630     #include <stdlib.h>
22631
22632         long labs(long i);
22632         long long llabs(long long i);
```

22633 DESCRIPTION

22634 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22635 conflict between the requirements described here and the ISO C standard is unintentional. This
22636 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

22637 The **labs()** function shall compute the absolute value of the **long** integer operand *i*. The **llabs()**
22638 function shall compute the absolute value of the **long long** integer operand *i*. If the result cannot
22639 be represented, the behavior is undefined.

22640 RETURN VALUE

22641 The **labs()** function shall return the absolute value of the **long** integer operand. The **labs()**
22642 function shall return the absolute value of the **long long** integer operand.

22643 ERRORS

22644 No errors are defined.

22645 EXAMPLES

22646 None.

22647 APPLICATION USAGE

22648 None.

22649 RATIONALE

22650 None.

22651 FUTURE DIRECTIONS

22652 None.

22653 SEE ALSO

22654 **abs()**, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

22655 CHANGE HISTORY

22656 First released in Issue 4. Derived from the ISO C standard.

22657 Issue 6

22658 The **llabs()** function is added for alignment with the ISO/IEC 9899:1999 standard.

22659 NAME

22660 lchown — change the owner and group of a symbolic link

22661 SYNOPSIS

22662 XSI #include <unistd.h>

22663 int lchown(const char *path, uid_t owner, gid_t group);

22664

22665 DESCRIPTION

22666 The *lchown()* function shall be equivalent to *chown()*, except in the case where the named file is a
22667 symbolic link. In this case, *lchown()* shall change the ownership of the symbolic link file itself,
22668 while *chown()* changes the ownership of the file or directory to which the symbolic link refers.

22669 RETURN VALUE

22670 Upon successful completion, *lchown()* shall return 0. Otherwise, it shall return -1 and set *errno* to
22671 indicate an error.

22672 ERRORS

22673 The *lchown()* function shall fail if:

- 22674 [EACCES] Search permission is denied on a component of the path prefix of *path*.
- 22675 [EINVAL] The owner or group ID is not a value supported by the implementation.
- 22676 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.
- 22677 [ENAMETOOLONG]
22678 The length of a pathname exceeds {PATH_MAX} or a pathname component is
22679 longer than {NAME_MAX}.
- 22680 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.
- 22681 [ENOTDIR] A component of the path prefix of *path* is not a directory.
- 22682 [EOPNOTSUPP] The *path* argument names a symbolic link and the implementation does not
22683 support setting the owner or group of a symbolic link.
- 22684 [EPERM] The effective user ID does not match the owner of the file and the process
22685 does not have appropriate privileges.
- 22686 [EROFS] The file resides on a read-only file system.

22687 The *lchown()* function may fail if:

- 22688 [EIO] An I/O error occurred while reading or writing to the file system.
- 22689 [EINTR] A signal was caught during execution of the function.
- 22690 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
22691 resolution of the *path* argument.
- 22692 [ENAMETOOLONG]
22693 Pathname resolution of a symbolic link produced an intermediate result
22694 whose length exceeds {PATH_MAX}.
- 22695

22696 EXAMPLES**22697 Changing the Current Owner of a File**

22698 The following example shows how to change the ownership of the symbolic link named
22699 **/modules/pass1** to the user ID associated with “jones” and the group ID associated with “cnd”.

22700 The numeric value for the user ID is obtained by using the *getpwnam()* function. The numeric
22701 value for the group ID is obtained by using the *getgrnam()* function.

```
22702 #include <sys/types.h>
22703 #include <unistd.h>
22704 #include <pwd.h>
22705 #include <grp.h>

22706 struct passwd *pwd;
22707 struct group *grp;
22708 char *path = "/modules/pass1";
22709 ...
22710 pwd = getpwnam("jones");
22711 grp = getgrnam("cnd");
22712 lchown(path, pwd->pw_uid, grp->gr_gid);
```

22713 APPLICATION USAGE

22714 On implementations which support symbolic links as directory entries rather than files, *lchown()*
22715 may fail.

22716 RATIONALE

22717 None.

22718 FUTURE DIRECTIONS

22719 None.

22720 SEE ALSO

22721 *chown()*, *symlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

22722 CHANGE HISTORY

22723 First released in Issue 4, Version 2.

22724 Issue 5

22725 Moved from X/OPEN UNIX extension to BASE.

22726 Issue 6

22727 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
22728 [ELOOP] error condition is added.

22729 The Open Group Base Resolution bwg2001-013 is applied, adding wording to the
22730 APPLICATION USAGE.

22731 NAME

22732 lcong48 — seed a uniformly distributed pseudo-random signed long integer generator

22733 SYNOPSIS

22734 XSI #include <stdlib.h>

22735 void lcong48(unsigned short *param*[7]);

22736

22737 DESCRIPTION

22738 Refer to *drand48()*.

22739 NAME

22740 ldexp, ldexpf, ldexpl — load exponent of a floating-point number

22741 SYNOPSIS

```
22742     #include <math.h>
22743
22744     double ldexp(double x, int exp);
22745     float ldexpf(float x, int exp);
22746     long double ldexpl(long double x, int exp);
```

22746 DESCRIPTION

22747 CX 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.

22750 These functions shall compute the quantity $x * 2^{exp}$.

22751 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.

22755 RETURN VALUE

22756 Upon successful completion, these functions shall return *x* multiplied by 2, raised to the power *exp*.

22758 If these functions would cause overflow, a range error shall occur and *ldexp()*, *ldexpf()*, and *ldexpl()* shall return $\pm\text{HUGE_VAL}$, $\pm\text{HUGE_VALF}$, and $\pm\text{HUGE_VALL}$ (according to the sign of *x*), respectively.

22761 If the correct value would cause underflow, and is not representable, a range error may occur, and either 0.0 (if supported), or an implementation-defined value shall be returned.

22763 MX If *x* is NaN, a NaN shall be returned.

22764 If *x* is ± 0 or $\pm\text{Inf}$, *x* shall be returned.

22765 If *exp* is 0, *x* shall be returned.

22766 If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

22768 ERRORS

22769 These functions shall fail if:

22770 Range Error The result overflows.

22771 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.

22775 These functions may fail if:

22776 Range Error The result underflows.

22777 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 underflow floating-point exception shall be raised.

22781 EXAMPLES

22782 None.

22783 APPLICATION USAGE

22784 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
22785 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

22786 RATIONALE

22787 None.

22788 FUTURE DIRECTIONS

22789 None.

22790 SEE ALSO

22791 *feclearexcept()*, *fetestexcept()*, *frexp()*, *isnan()*, the Base Definitions volume of
22792 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
22793 *<math.h>*

22794 CHANGE HISTORY

22795 First released in Issue 1. Derived from Issue 1 of the SVID.

22796 Issue 5

22797 The DESCRIPTION is updated to indicate how an application should check for an error. This
22798 text was previously published in the APPLICATION USAGE section.

22799 Issue 6

22800 The *Idexpf()* and *Idexpl()* functions are added for alignment with the ISO/IEC 9899:1999
22801 standard.

22802 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
22803 revised to align with the ISO/IEC 9899:1999 standard.

22804 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
22805 marked.

22806 NAME

22807 ldiv, lldiv — compute quotient and remainder of a long division

22808 SYNOPSIS

```
22809 #include <stdlib.h>
22810 ldiv_t ldiv(long numer, long denom);
22811 lldiv_t lldiv(long long numer, long long denom);
```

22812 DESCRIPTION

22813 CX The functionality described on this reference page is aligned with the ISO C standard. Any
22814 conflict between the requirements described here and the ISO C standard is unintentional. This
22815 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

22816 These functions shall compute the quotient and remainder of the division of the numerator
22817 *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the **long**
22818 integer (for the *ldiv()* function) or **long long** integer (for the *lldiv()* function) of lesser magnitude
22819 that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is
22820 undefined; otherwise, *quot* * *denom*+*rem* shall equal *numer*.

22821 RETURN VALUE

22822 The *ldiv()* function shall return a structure of type **ldiv_t**, comprising both the quotient and the
22823 remainder. The structure shall include the following members, in any order:

```
22824 long quot; /* Quotient */
22825 long rem; /* Remainder */
```

22826 The *lldiv()* function shall return a structure of type **lldiv_t**, comprising both the quotient and the
22827 remainder. The structure shall include the following members, in any order:

```
22828 long long quot; /* Quotient */
22829 long long rem; /* Remainder */
```

22830 ERRORS

22831 No errors are defined.

22832 EXAMPLES

22833 None.

22834 APPLICATION USAGE

22835 None.

22836 RATIONALE

22837 None.

22838 FUTURE DIRECTIONS

22839 None.

22840 SEE ALSO

22841 *div()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

22842 CHANGE HISTORY

22843 First released in Issue 4. Derived from the ISO C standard.

22844 Issue 6

22845 The *lldiv()* function is added for alignment with the ISO/IEC 9899:1999 standard.

22846 NAME

22847 lfind — find entry in a linear search table

22848 SYNOPSIS

22849 XSI #include <search.h>

22850 void *lfind(const void *key, const void *base, size_t *nelp,
22851 size_t width, int (*compar)(const void *, const void *));
22852

22853 DESCRIPTION

22854 Refer to *lsearch()*.

22855 NAME

22856 lgamma, lgammaf, lgammal — log gamma function

22857 SYNOPSIS

```
22858     #include <math.h>
22859
22860     double lgamma(double x);
22861     float lgammaf(float x);
22862     long double lgammal(long double x);
22863     extern int signgam;
```

22864 DESCRIPTION

22865 CX 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.

22868 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
 22869 need not be a non-positive integer ($\Gamma(x)$ is defined over the reals, except the non-positive
 22870 integers).

22872 XSI The sign of $\Gamma(x)$ is returned in the external integer *signgam*.

22873 CX These functions need not be reentrant. A function that is not required to be reentrant is not
 22874 required to be thread-safe.

22875 An application wishing to check for error situations should set *errno* to zero and call
 22876 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
 22877 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
 22878 zero, an error has occurred.

22879 RETURN VALUE

22880 Upon successful completion, these functions shall return the logarithmic gamma of x .

22881 If x is a non-positive integer, a pole error shall occur and *lgamma()*, *lgammaf()*, and *lgammal()*
 22882 shall return *+HUGE_VAL*, *+HUGE_VALF*, and *+HUGE_VALL*, respectively.

22883 If the correct value would cause overflow, a range error shall occur and *lgamma()*, *lgammaf()*, and
 22884 *lgammal()* shall return *±HUGE_VAL*, *±HUGE_VALF*, and *±HUGE_VALL* (having the same
 22885 sign as the correct value), respectively.

22886 MX If x is NaN, a NaN shall be returned.

22887 If x is 1 or 2, +0 shall be returned.

22888 If x is ±Inf, +Inf shall be returned.

22889 ERRORS

22890 These functions shall fail if:

22891 Pole Error The x argument is a negative integer or zero.

22892 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 22893 then *errno* shall be set to [ERANGE]. If the integer expression
 22894 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-
 22895 zero floating-point exception shall be raised.

22896 Range Error The result overflows.

22897 If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
22898 then *errno* shall be set to [ERANGE]. If the integer expression
22899 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow
22900 floating-point exception shall be raised.

22901 EXAMPLES

22902 None.

22903 APPLICATION USAGE

22904 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
22905 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

22906 RATIONALE

22907 None.

22908 FUTURE DIRECTIONS

22909 None.

22910 SEE ALSO

22911 *exp()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
22912 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

22913 CHANGE HISTORY

22914 First released in Issue 3.

22915 Issue 5

22916 The DESCRIPTION is updated to indicate how an application should check for an error. This
22917 text was previously published in the APPLICATION USAGE section.

22918 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

22919 Issue 6

22920 The *lgamma()* function is no longer marked as an extension.

22921 The *lgammaf()* and *lgammal()* functions are added for alignment with the ISO/IEC 9899:1999
22922 standard.

22923 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
22924 revised to align with the ISO/IEC 9899:1999 standard.

22925 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
22926 marked.

22927 XSI extensions are marked.

22928 NAME

22929 link — link to a file

22930 SYNOPSIS

22931 #include <unistd.h>
22932 int link(const char *path1, const char *path2);

22933 DESCRIPTION

22934 The *link()* function shall create a new link (directory entry) for the existing file, *path1*.22935 The *path1* argument points to a pathname naming an existing file. The *path2* argument points to
22936 a pathname naming the new directory entry to be created. The *link()* function shall atomically
22937 create a new link for the existing file and the link count of the file shall be incremented by one.22938 If *path1* names a directory, *link()* shall fail unless the process has appropriate privileges and the
22939 implementation supports using *link()* on directories.22940 Upon successful completion, *link()* shall mark for update the *st_ctime* field of the file. Also, the
22941 *st_ctime* and *st_mtime* fields of the directory that contains the new entry shall be marked for
22942 update.22943 If *link()* fails, no link shall be created and the link count of the file shall remain unchanged.22944 The implementation may require that the calling process has permission to access the existing
22945 file.

22946 RETURN VALUE

22947 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
22948 indicate the error.

22949 ERRORS

22950 The *link()* function shall fail if:22951 [EACCES] A component of either path prefix denies search permission, or the requested
22952 link requires writing in a directory that denies write permission, or the calling
22953 process does not have permission to access the existing file and this is
22954 required by the implementation.22955 [EEXIST] The *path2* argument resolves to an existing file or refers to a symbolic link.22956 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path1* or
22957 *path2* argument.22958 [EMLINK] The number of links to the file named by *path1* would exceed {LINK_MAX}.22959 [ENAMETOOLONG] The length of the *path1* or *path2* argument exceeds {PATH_MAX} or a
22960 pathname component is longer than {NAME_MAX}.22962 [ENOENT] A component of either path prefix does not exist; the file named by *path1* does
22963 not exist; or *path1* or *path2* points to an empty string.

22964 [ENOSPC] The directory to contain the link cannot be extended.

22965 [ENOTDIR] A component of either path prefix is not a directory.

22966 [EPERM] The file named by *path1* is a directory and either the calling process does not
22967 have appropriate privileges or the implementation prohibits using *link()* on
22968 directories.

22969 [EROFS] The requested link requires writing in a directory on a read-only file system.
22970 [EXDEV] The link named by *path2* and the file named by *path1* are on different file
22971 systems and the implementation does not support links between file systems.
22972 XSR [EXDEV] *path1* refers to a named STREAM.
22973 The *link()* function may fail if:
22974 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
22975 resolution of the *path1* or *path2* argument.
22976 [ENAMETOOLONG]
22977 As a result of encountering a symbolic link in resolution of the *path1* or *path2*
22978 argument, the length of the substituted pathname string exceeded
22979 {PATH_MAX}.

22980 EXAMPLES

22981 Creating a Link to a File

22982 The following example shows how to create a link to a file named **/home/cnd/mod1** by creating a
22983 new directory entry named **/modules/pass1**.

```
22984 #include <unistd.h>  
22985 char *path1 = "/home/cnd/mod1";  
22986 char *path2 = "/modules/pass1";  
22987 int status;  
22988 ...  
22989 status = link (path1, path2);
```

22990 Creating a Link to a File Within a Program

22991 In the following program example, the *link()* function links the **/etc/passwd** file (defined as
22992 **PASSWDFILE**) to a file named **/etc/opasswd** (defined as **SAVEFILE**), which is used to save the
22993 current password file. Then, after removing the current password file (defined as
22994 **PASSWDFILE**), the new password file is saved as the current password file using the *link()*
22995 function again.

```
22996 #include <unistd.h>  
22997 #define LOCKFILE "/etc/ptmp"  
22998 #define PASSWDFILE "/etc/passwd"  
22999 #define SAVEFILE "/etc/opasswd"  
23000 ...  
23001 /* Save current password file */  
23002 link (PASSWDFILE, SAVEFILE);  
23003 /* Remove current password file. */  
23004 unlink (PASSWDFILE);  
23005 /* Save new password file as current password file. */  
23006 link (LOCKFILE,PASSWDFILE);
```

23007 APPLICATION USAGE

23008 Some implementations do allow links between file systems.

23009 RATIONALE

23010 Linking to a directory is restricted to the superuser in most historical implementations because
23011 this capability may produce loops in the file hierarchy or otherwise corrupt the file system. This
23012 volume of IEEE Std 1003.1-2001 continues that philosophy by prohibiting *link()* and *unlink()*
23013 from doing this. Other functions could do it if the implementor designed such an extension.

23014 Some historical implementations allow linking of files on different file systems. Wording was
23015 added to explicitly allow this optional behavior.

23016 The exception for cross-file system links is intended to apply only to links that are
23017 programmatically indistinguishable from ‘hard’ links.

23018 FUTURE DIRECTIONS

23019 None.

23020 SEE ALSO

23021 *symlink()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

23022 CHANGE HISTORY

23023 First released in Issue 1. Derived from Issue 1 of the SVID.

23024 Issue 6

23025 The following new requirements on POSIX implementations derive from alignment with the
23026 Single UNIX Specification:

- 23027 • The [ELOOP] mandatory error condition is added.
- 23028 • A second [ENAMETOOLONG] is added as an optional error condition.

23029 The following changes were made to align with the IEEE P1003.1a draft standard:

- 23030 • An explanation is added of the action when *path2* refers to a symbolic link.
- 23031 • The [ELOOP] optional error condition is added.

23032 NAME

23033 lio_listio — list directed I/O (REALTIME)

23034 SYNOPSIS

23035 AIO #include <aio.h>

```
23036 int lio_listio(int mode, struct aiocb *restrict const list[restrict],
23037     int nent, struct sigevent *restrict sig);
```

23038

23039 DESCRIPTION

23040 The *lio_listio()* function shall initiate a list of I/O requests with a single function call.

23041 The *mode* argument takes one of the values LIO_WAIT or LIO_NOWAIT declared in <aio.h> and
 23042 determines whether the function returns when the I/O operations have been completed, or as
 23043 soon as the operations have been queued. If the *mode* argument is LIO_WAIT, the function shall
 23044 wait until all I/O is complete and the *sig* argument shall be ignored.

23045 If the *mode* argument is LIO_NOWAIT, the function shall return immediately, and asynchronous
 23046 notification shall occur, according to the *sig* argument, when all the I/O operations complete. If
 23047 *sig* is NULL, then no asynchronous notification shall occur. If *sig* is not NULL, asynchronous
 23048 notification occurs as specified in Section 2.4.1 (on page 28) when all the requests in *list* have
 23049 completed.

23050 The I/O requests enumerated by *list* are submitted in an unspecified order.23051 The *list* argument is an array of pointers to **aiocb** structures. The array contains *nent* elements.
 23052 The array may contain NULL elements, which shall be ignored.

23053 If the buffer pointed to by *list* or the **aiocb** structures pointed to by the elements of the array *list* 2
 23054 become illegal addresses before all asynchronous I/O completed and, if necessary, the 2
 23055 notification is sent, then the behavior is undefined. If the buffers pointed to by the *aio_buf* 2
 23056 member of the **aiocb** structure pointed to by the elements of the array *list* become illegal 2
 23057 addresses prior to the asynchronous I/O associated with that **aiocb** structure being completed, 2
 23058 the behavior is undefined. 2

23059 The *aio_lio_opcode* field of each **aiocb** structure specifies the operation to be performed. The 23060 supported operations are LIO_READ, LIO_WRITE, and LIO_NOP; these symbols are defined in
 23061 <aio.h>. The LIO_NOP operation causes the list entry to be ignored. If the *aio_lio_opcode* 23062 element is equal to LIO_READ, then an I/O operation is submitted as if by a call to *aio_read()* 23063 with the *aiocbp* equal to the address of the **aiocb** structure. If the *aio_lio_opcode* element is 23064 equal to LIO_WRITE, then an I/O operation is submitted as if by a call to *aio_write()* with the *aiocbp* 23065 equal to the address of the **aiocb** structure.

23066 The *aio_fildes* member specifies the file descriptor on which the operation is to be performed.23067 The *aio_buf* member specifies the address of the buffer to or from which the data is transferred.23068 The *aio_nbytes* member specifies the number of bytes of data to be transferred.

23069 The members of the **aiocb** structure further describe the I/O operation to be performed, in a 23070 manner identical to that of the corresponding **aiocb** structure when used by the *aio_read()* and 23071 *aio_write()* functions.

23072 The *nent* argument specifies how many elements are members of the list; that is, the length of the 23073 array.

23074 The behavior of this function is altered according to the definitions of synchronized I/O data 23075 integrity completion and synchronized I/O file integrity completion if synchronized I/O is 23076 enabled on the file associated with *aio_fildes*.

23077	For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with <i>aiocbp->aio_fildes</i> .	
23079	If <i>sig->sigev_notify</i> is SIGEV_THREAD and <i>sig->sigev_notify_attributes</i> is a non-NULL pointer and the block pointed to by this pointer becomes an illegal address prior to all asynchronous I/O being completed, then the behavior is undefined.	2
23080		2
23081		2

23082 **RETURN VALUE**

23083 If the *mode* argument has the value LIO_NOWAIT, the *lio_listio()* function shall return the value
 23084 zero if the I/O operations are successfully queued; otherwise, the function shall return the value
 23085 -1 and set *errno* to indicate the error.

23086 If the *mode* argument has the value LIO_WAIT, the *lio_listio()* function shall return the value
 23087 zero when all the indicated I/O has completed successfully. Otherwise, *lio_listio()* shall return a
 23088 value of -1 and set *errno* to indicate the error.

23089 In either case, the return value only indicates the success or failure of the *lio_listio()* call itself,
 23090 not the status of the individual I/O requests. In some cases one or more of the I/O requests
 23091 contained in the list may fail. Failure of an individual request does not prevent completion of
 23092 any other individual request. To determine the outcome of each I/O request, the application
 23093 shall examine the error status associated with each **aiocb** control block. The error statuses so
 23094 returned are identical to those returned as the result of an *aio_read()* or *aio_write()* function.

23095 **ERRORS**

23096 The *lio_listio()* function shall fail if:

23097	[EAGAIN]	The resources necessary to queue all the I/O requests were not available. The application may check the error status for each aiocb to determine the individual request(s) that failed.
23100	[EAGAIN]	The number of entries indicated by <i>nent</i> would cause the system-wide limit {AIO_MAX} to be exceeded.
23102	[EINVAL]	The <i>mode</i> argument is not a proper value, or the value of <i>nent</i> was greater than {AIO_LISTIO_MAX}.
23104	[EINTR]	A signal was delivered while waiting for all I/O requests to complete during an LIO_WAIT operation. Note that, since each I/O operation invoked by <i>lio_listio()</i> may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited. Outstanding I/O requests are not canceled, and the application shall examine each list element to determine whether the request was initiated, canceled, or completed.
23111	[EIO]	One or more of the individual I/O operations failed. The application may check the error status for each aiocb structure to determine the individual request(s) that failed.

23114 In addition to the errors returned by the *lio_listio()* function, if the *lio_listio()* function succeeds
 23115 or fails with errors of [EAGAIN], [EINTR], or [EIO], then some of the I/O specified by the list
 23116 may have been initiated. If the *lio_listio()* function fails with an error code other than [EAGAIN],
 23117 [EINTR], or [EIO], no operations from the list shall have been initiated. The I/O operation
 23118 indicated by each list element can encounter errors specific to the individual read or write
 23119 function being performed. In this event, the error status for each **aiocb** control block contains the
 23120 associated error code. The error codes that can be set are the same as would be set by a *read()* or
 23121 *write()* function, with the following additional error codes possible:

23122	[EAGAIN]	The requested I/O operation was not queued due to resource limitations.
23123	[ECANCELED]	The requested I/O was canceled before the I/O completed due to an explicit <i>aio_cancel()</i> request.
23125	[EFBIG]	The <i>aiocbp->aio_lio_opcode</i> is LIO_WRITE, the file is a regular file, <i>aiocbp->aio_nbytes</i> is greater than 0, and the <i>aiocbp->aio_offset</i> is greater than or equal to the offset maximum in the open file description associated with <i>aiocbp->aio_fildes</i> .
23129	[EINPROGRESS]	The requested I/O is in progress.
23130	[EOVERFLOW]	The <i>aiocbp->aio_lio_opcode</i> is LIO_READ, the file is a regular file, <i>aiocbp->aio_nbytes</i> is greater than 0, and the <i>aiocbp->aio_offset</i> is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with <i>aiocbp->aio_fildes</i> .

23134 EXAMPLES

23135 None.

23136 APPLICATION USAGE

23137 None.

23138 RATIONALE

23139 Although it may appear that there are inconsistencies in the specified circumstances for error codes, the [EIO] error condition applies when any circumstance relating to an individual operation makes that operation fail. This might be due to a badly formulated request (for example, the *aio_lio_opcode* field is invalid, and *aio_error()* returns [EINVAL]) or might arise from application behavior (for example, the file descriptor is closed before the operation is initiated, and *aio_error()* returns [EBADF]).

23145 The limitation on the set of error codes returned when operations from the list shall have been initiated enables applications to know when operations have been started and whether *aio_error()* is valid for a specific operation.

23148 FUTURE DIRECTIONS

23149 None.

23150 SEE ALSO

23151 *aio_read()*, *aio_write()*, *aio_error()*, *aio_return()*, *aio_cancel()*, *close()*, *exec*, *exit()*, *fork()*, *lseek()*,
23152 *read()*, the Base Definitions volume of IEEE Std 1003.1-2001, <*aio.h*>

23153 CHANGE HISTORY

23154 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

23155 Large File Summit extensions are added.

23156 Issue 6

23157 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.

23159 The *lio_listio()* function is marked as part of the Asynchronous Input and Output option.

23160 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- 23162 • In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs
23163 past the offset maximum established in the open file description associated with
23164 *aiocbp->aio_fildes*. This change is to support large files.

23165	• The [EBIG] and [EOVERFLOW] error conditions are defined. This change is to support large files.	
23166		
23167	The DESCRIPTION is updated to avoid use of the term “must” for application requirements.	
23168	The restrict keyword is added to the <i>lio_listio()</i> prototype for alignment with the ISO/IEC 9899:1999 standard.	
23169		
23170	Issue 6	2
23171	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/53 is applied, adding new text for symmetry with the <i>aio_read()</i> and <i>aio_write()</i> functions to the DESCRIPTION.	2
23172		2
23173	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/54 is applied, adding text to the DESCRIPTION making it explicit that the user is required to keep the structure pointed to by <i>sig->sigev_notify_attributes</i> valid until the last asynchronous operation finished and the notification has been sent.	2
23174		2
23175		2
23176		2

23177 NAME

23178 listen — listen for socket connections and limit the queue of incoming connections

23179 SYNOPSIS

```
23180        #include <sys/socket.h>
23181        int listen(int socket, int backlog);
```

23182 DESCRIPTION

23183 The *listen()* function shall mark a connection-mode socket, specified by the *socket* argument, as
23184 accepting connections.

23185 The *backlog* argument provides a hint to the implementation which the implementation shall use
23186 to limit the number of outstanding connections in the socket's listen queue. Implementations
23187 may impose a limit on *backlog* and silently reduce the specified value. Normally, a larger *backlog*
23188 argument value shall result in a larger or equal length of the listen queue. Implementations shall
23189 support values of *backlog* up to SOMAXCONN, defined in <sys/socket.h>.

23190 The implementation may include incomplete connections in its listen queue. The limits on the
23191 number of incomplete connections and completed connections queued may be different.

23192 The implementation may have an upper limit on the length of the listen queue—either global or
23193 per accepting socket. If *backlog* exceeds this limit, the length of the listen queue is set to the limit.

23194 If *listen()* is called with a *backlog* argument value that is less than 0, the function behaves as if it
23195 had been called with a *backlog* argument value of 0.

23196 A *backlog* argument of 0 may allow the socket to accept connections, in which case the length of
23197 the listen queue may be set to an implementation-defined minimum value.

23198 The socket in use may require the process to have appropriate privileges to use the *listen()*
23199 function.

23200 RETURN VALUE

23201 Upon successful completions, *listen()* shall return 0; otherwise, -1 shall be returned and *errno* set
23202 to indicate the error.

23203 ERRORS

23204 The *listen()* function shall fail if:

23205 [EBADF] The *socket* argument is not a valid file descriptor.

23206 [EDESTADDRREQ] The socket is not bound to a local address, and the protocol does not support
23207 listening on an unbound socket.

23209 [EINVAL] The *socket* is already connected.

23210 [ENOTSOCK] The *socket* argument does not refer to a socket.

23211 [EOPNOTSUPP] The socket protocol does not support *listen()*.

23212 The *listen()* function may fail if:

23213 [EACCES] The calling process does not have the appropriate privileges.

23214 [EINVAL] The *socket* has been shut down.

23215 [ENOBUFS] Insufficient resources are available in the system to complete the call.

23216 EXAMPLES

23217 None.

23218 APPLICATION USAGE

23219 None.

23220 RATIONALE

23221 None.

23222 FUTURE DIRECTIONS

23223 None.

23224 SEE ALSO

23225 *accept()*, *connect()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

23226 CHANGE HISTORY

23227 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

23228 The DESCRIPTION is updated to describe the relationship of SOMAXCONN and the *backlog* argument.
23229

```
23230 NAME
23231     llabs — return a long integer absolute value
23232 SYNOPSIS
23233     #include <stdlib.h>
23234     long long llabs(long long i);
23235 DESCRIPTION
23236     Refer to labs().
```

23237 NAME

23238 **lldiv** — compute quotient and remainder of a long division

23239 SYNOPSIS

23240 #include <stdlib.h>

23241 lldiv_t lldiv(long long *numer*, long long *denom*) ;

23242 DESCRIPTION

23243 Refer to *ldiv()*.

23244 NAME

23245 llrint, llrintf, llrintl — round to the nearest integer value using current rounding direction

23246 SYNOPSIS

```
23247 #include <math.h>
23248 long long llrint(double x);
23249 long long llrintf(float x);
23250 long long llrintl(long double x);
```

23251 DESCRIPTION

23252 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23253 conflict between the requirements described here and the ISO C standard is unintentional. This
23254 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23255 These functions shall round their argument to the nearest integer value, rounding according to
23256 the current rounding direction.

23257 An application wishing to check for error situations should set *errno* to zero and call
23258 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
23259 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
23260 zero, an error has occurred.

23261 RETURN VALUE

23262 Upon successful completion, these functions shall return the rounded integer value.

23263 MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.

23264 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

23265 If *x* is -Inf, a domain error shall occur and an unspecified value is returned.

23266 If the correct value is positive and too large to represent as a **long long**, a domain error shall
23267 occur and an unspecified value is returned.

23268 If the correct value is negative and too large to represent as a **long long**, a domain error shall
23269 occur and an unspecified value is returned.

23270 ERRORS

23271 These functions shall fail if:

23272 MX Domain Error The *x* argument is NaN or ±Inf, or the correct value is not representable as an
23273 integer.

23274 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
23275 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
23276 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
23277 shall be raised.

23278 EXAMPLES

23279 None.

23280 APPLICATION USAGE

23281 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
23282 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23283 RATIONALE

23284 These functions provide floating-to-integer conversions. They round according to the current
23285 rounding direction. If the rounded value is outside the range of the return type, the numeric
23286 result is unspecified and the invalid floating-point exception is raised. When they raise no other
23287 floating-point exception and the result differs from the argument, they raise the inexact

23288 floating-point exception.

23289 **FUTURE DIRECTIONS**

23290 None.

23291 **SEE ALSO**

23292 *feclearexcept()*, *fetestexcept()*, *lrint()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section
23293 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

23294 **CHANGE HISTORY**

23295 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

23296 NAME

23297 llround, llroundf, llroundl — round to nearest integer value

23298 SYNOPSIS

```
23299 #include <math.h>
23300 long long llround(double x);
23301 long long llroundf(float x);
23302 long long llroundl(long double x);
```

23303 DESCRIPTION

23304 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 23305 conflict between the requirements described here and the ISO C standard is unintentional. This
 23306 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23307 These functions shall round their argument to the nearest integer value, rounding halfway cases
 23308 away from zero, regardless of the current rounding direction.

23309 An application wishing to check for error situations should set *errno* to zero and call
 23310 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
 23311 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
 23312 zero, an error has occurred.

23313 RETURN VALUE

23314 Upon successful completion, these functions shall return the rounded integer value.

23315 MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.

23316 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

23317 If *x* is -Inf, a domain error shall occur and an unspecified value is returned.

23318 If the correct value is positive and too large to represent as a **long long**, a domain error shall
 23319 occur and an unspecified value is returned.

23320 If the correct value is negative and too large to represent as a **long long**, a domain error shall
 23321 occur and an unspecified value is returned.

23322 ERRORS

23323 These functions shall fail if:

23324 MX Domain Error The *x* argument is NaN or ±Inf, or the correct value is not representable as an
 23325 integer.

23326 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 23327 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 23328 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 23329 shall be raised.

23330 EXAMPLES

23331 None.

23332 APPLICATION USAGE

23333 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
 23334 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23335 RATIONALE

23336 These functions differ from the *llrint()* functions in that the default rounding direction for the
 23337 *llround()* functions round halfway cases away from zero and need not raise the inexact floating-
 23338 point exception for non-integer arguments that round to within the range of the return type.

23339 **FUTURE DIRECTIONS**

23340 None.

23341 **SEE ALSO**

23342 *feclearexcept()*, *fetestexcept()*, *lround()*, the Base Definitions volume of IEEE Std 1003.1-2001,
23343 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

23344 **CHANGE HISTORY**

23345 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

23346 NAME

23347 **localeconv** — return locale-specific information

23348 SYNOPSIS

23349

```
#include <locale.h>
```


23350

```
struct lconv *localeconv(void);
```

23351 DESCRIPTION

23352 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23353 conflict between the requirements described here and the ISO C standard is unintentional. This
23354 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.23355 The **localeconv()** function shall set the components of an object with the type **struct lconv** with
23356 the values appropriate for the formatting of numeric quantities (monetary and otherwise)
23357 according to the rules of the current locale.23358 The members of the structure with type **char *** are pointers to strings, any of which (except
23359 **decimal_point**) can point to " ", to indicate that the value is not available in the current locale or
23360 is of zero length. The members with type **char** are non-negative numbers, any of which can be
23361 **{CHAR_MAX}** to indicate that the value is not available in the current locale.

23362 The members include the following:

23363 **char *decimal_point**

23364 The radix character used to format non-monetary quantities.

23365 **char *thousands_sep**23366 The character used to separate groups of digits before the decimal-point character in
23367 formatted non-monetary quantities.23368 **char *grouping**23369 A string whose elements taken as one-byte integer values indicate the size of each group of
23370 digits in formatted non-monetary quantities.23371 **char *int_curr_symbol**23372 The international currency symbol applicable to the current locale. The first three
23373 characters contain the alphabetic international currency symbol in accordance with those
23374 specified in the ISO 4217:2001 standard. The fourth character (immediately preceding the
23375 null byte) is the character used to separate the international currency symbol from the
23376 monetary quantity.1
123377 **char *currency_symbol**

23378 The local currency symbol applicable to the current locale.

23379 **char *mon_decimal_point**

23380 The radix character used to format monetary quantities.

23381 **char *mon_thousands_sep**23382 The separator for groups of digits before the decimal-point in formatted monetary
23383 quantities.23384 **char *mon_grouping**23385 A string whose elements taken as one-byte integer values indicate the size of each group of
23386 digits in formatted monetary quantities.23387 **char *positive_sign**

23388 The string used to indicate a non-negative valued formatted monetary quantity.

23389 **char *negative_sign**
23390 The string used to indicate a negative valued formatted monetary quantity.

23391 **char int_frac_digits**
23392 The number of fractional digits (those after the decimal-point) to be displayed in an
23393 internationally formatted monetary quantity.

23394 **char frac_digits**
23395 The number of fractional digits (those after the decimal-point) to be displayed in a
23396 formatted monetary quantity.

23397 **char p_cs_precedes**
23398 Set to 1 if the **currency_symbol** precedes the value for a non-negative formatted monetary 1
23399 quantity. Set to 0 if the symbol succeeds the value.

23400 **char p_sep_by_space**
23401 Set to a value indicating the separation of the **currency_symbol**, the sign string, and the 1
23402 value for a non-negative formatted monetary quantity.

23403 **char n_cs_precedes**
23404 Set to 1 if the **currency_symbol** precedes the value for a negative formatted monetary 1
23405 quantity. Set to 0 if the symbol succeeds the value.

23406 **char n_sep_by_space**
23407 Set to a value indicating the separation of the **currency_symbol**, the sign string, and the 1
23408 value for a negative formatted monetary quantity.

23409 **char p_sign_posn**
23410 Set to a value indicating the positioning of the **positive_sign** for a non-negative formatted
23411 monetary quantity.

23412 **char n_sign_posn**
23413 Set to a value indicating the positioning of the **negative_sign** for a negative formatted
23414 monetary quantity.

23415 **char int_p_cs_precedes**
23416 Set to 1 or 0 if the **int_curr_symbol** respectively precedes or succeeds the value for a non-
23417 negative internationally formatted monetary quantity.

23418 **char int_n_cs_precedes**
23419 Set to 1 or 0 if the **int_curr_symbol** respectively precedes or succeeds the value for a
23420 negative internationally formatted monetary quantity.

23421 **char int_p_sep_by_space**
23422 Set to a value indicating the separation of the **int_curr_symbol**, the sign string, and the
23423 value for a non-negative internationally formatted monetary quantity.

23424 **char int_n_sep_by_space**
23425 Set to a value indicating the separation of the **int_curr_symbol**, the sign string, and the
23426 value for a negative internationally formatted monetary quantity.

23427 **char int_p_sign_posn**
23428 Set to a value indicating the positioning of the **positive_sign** for a non-negative
23429 internationally formatted monetary quantity.

23430 **char int_n_sign_posn**
23431 Set to a value indicating the positioning of the **negative_sign** for a negative internationally
23432 formatted monetary quantity.

23433 The elements of **grouping** and **mon_grouping** are interpreted according to the following:

23434 {CHAR_MAX} No further grouping is to be performed.

23435 0 The previous element is to be repeatedly used for the remainder of the digits.

23436 *other* The integer value is the number of digits that comprise the current group. The
23437 next element is examined to determine the size of the next group of digits
23438 before the current group.

23439 The values of **p_sep_by_space**, **n_sep_by_space**, **int_p_sep_by_space**, and **int_n_sep_by_space**
23440 are interpreted according to the following:

23441 0 No space separates the currency symbol and value.

23442 1 If the currency symbol and sign string are adjacent, a space separates them from the value;
23443 otherwise, a space separates the currency symbol from the value.

23444 2 If the currency symbol and sign string are adjacent, a space separates them; otherwise, a
23445 space separates the sign string from the value.

23446 For **int_p_sep_by_space** and **int_n_sep_by_space**, the fourth character of **int_curr_symbol** is
23447 used instead of a space.

23448 The values of **p_sign_posn**, **n_sign_posn**, **int_p_sign_posn**, and **int_n_sign_posn** are
23449 interpreted according to the following:

23450 0 Parentheses surround the quantity and **currency_symbol** or **int_curr_symbol**.

23451 1 The sign string precedes the quantity and **currency_symbol** or **int_curr_symbol**.

23452 2 The sign string succeeds the quantity and **currency_symbol** or **int_curr_symbol**.

23453 3 The sign string immediately precedes the **currency_symbol** or **int_curr_symbol**.

23454 4 The sign string immediately succeeds the **currency_symbol** or **int_curr_symbol**.

23455 The implementation shall behave as if no function in this volume of IEEE Std 1003.1-2001 calls
23456 **localeconv()**.

23457 CX The **localeconv()** function need not be reentrant. A function that is not required to be reentrant is
23458 not required to be thread-safe.

23459 **RETURN VALUE**

23460 The **localeconv()** function shall return a pointer to the filled-in object. The application shall not
23461 modify the structure pointed to by the return value which may be overwritten by a subsequent
23462 call to **localeconv()**. In addition, calls to **setlocale()** with the categories **LC_ALL**, **LC_MONETARY**,
23463 or **LC_NUMERIC** may overwrite the contents of the structure.

23464 **ERRORS**

23465 No errors are defined.

23466 **EXAMPLES**

23467 None.

23468 **APPLICATION USAGE**

23469 The following table illustrates the rules which may be used by four countries to format monetary
23470 quantities.

23471

23472

23473

23474

23475

23476

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	SFr.1,234.56	SFr.1,234.56C	CHF 1,234.56

23477

For these four countries, the respective values for the monetary members of the structure returned by *localeconv()* are:

23479

23480

23481

23482

23483

23484

23485

23486

23487

23488

23489

23490

23491

23492

23493

23494

23495

23496

23497

23498

23499

23500

	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

23501 RATIONALE

23502 None.

23503 FUTURE DIRECTIONS

23504 None.

23505 SEE ALSO

23506 *isalpha()*, *isascii()*, *nl_langinfo()*, *printf()*, *scanf()*, *setlocale()*, *strcat()*, *strchr()*, *strcmp()*, *strcoll()*,
 23507 *strcpy()*, *strftime()*, *strlen()*, *strpbrk()*, *strspn()*, *strtok()*, *strxfrm()*, *strtod()*, the Base Definitions
 23508 volume of IEEE Std 1003.1-2001, *<langinfo.h>*, *<locale.h>*

23509 CHANGE HISTORY

23510 First released in Issue 4. Derived from the ANSI C standard.

23511 Issue 6

23512 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

23513 The RETURN VALUE section is rewritten to avoid use of the term "must".

23514 This reference page is updated for alignment with the ISO/IEC 9899:1999 standard.

23515 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

23516 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/31 is applied, removing references to 1
23517 **int_curr_symbol** and updating the descriptions of **p_sep_by_space** and **n_sep_by_space**. These 1
23518 changes are for alignment with the ISO C standard. 1

23519 NAME

23520 localtime, localtime_r — convert a time value to a broken-down local time

23521 SYNOPSIS

```
23522 #include <time.h>
23523 struct tm *localtime(const time_t *timer);
23524 TSF struct tm *localtime_r(const time_t *restrict timer,
23525         struct tm *restrict result);
23526
```

23527 DESCRIPTION

23528 CX For *localtime()*: The functionality described on this reference page is aligned with the ISO C
 23529 standard. Any conflict between the requirements described here and the ISO C standard is
 23530 unintentional. This volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23531 The *localtime()* function shall convert the time in seconds since the Epoch pointed to by *timer*
 23532 into a broken-down time, expressed as a local time. The function corrects for the timezone and
 23533 CX any seasonal time adjustments. Local timezone information is used as though *localtime()* calls
 23534 *tzset()*.

23535 The relationship between a time in seconds since the Epoch used as an argument to *localtime()*
 23536 and the **tm** structure (defined in the **<time.h>** header) is that the result shall be as specified in the
 23537 expression given in the definition of seconds since the Epoch (see the Base Definitions volume of
 23538 IEEE Std 1003.1-2001, Section 4.14, Seconds Since the Epoch) corrected for timezone and any
 23539 seasonal time adjustments, where the names in the structure and in the expression correspond.

23540 TSF The same relationship shall apply for *localtime_r()*.

23541 CX The *localtime()* function need not be reentrant. A function that is not required to be reentrant is
 23542 not required to be thread-safe.

23543 The *asctime()*, *ctime()*, *gmtime()*, and *localtime()* functions shall return values in one of two static
 23544 objects: a broken-down time structure and an array of type **char**. Execution of any of the
 23545 functions may overwrite the information returned in either of these objects by any of the other
 23546 functions.

23547 TSF The *localtime_r()* function shall convert the time in seconds since the Epoch pointed to by *timer*
 23548 into a broken-down time stored in the structure to which *result* points. The *localtime_r()* function
 23549 shall also return a pointer to that same structure.

23550 Unlike *localtime()*, the reentrant version is not required to set *tzname*.

23551 TSF If the reentrant version does not set *tzname*, it shall not set *daylight* and shall not set *timezone*.

2

23552 RETURN VALUE

23553 Upon successful completion, the *localtime()* function shall return a pointer to the broken-down
 23554 CX time structure. If an error is detected, *localtime()* shall return a null pointer and set *errno* to
 23555 indicate the error.

1

1

1

23556 TSF Upon successful completion, *localtime_r()* shall return a pointer to the structure pointed to by
 23557 the argument *result*. If an error is detected, *localtime_r()* shall return a null pointer and set *errno*
 23558 to indicate the error.

2

2

23559 ERRORS

23560 TSF The *localtime()* and *localtime_r()* functions shall fail if:

2

23561 CX [EOVERFLOW] The result cannot be represented.

1

23562 EXAMPLES

23563 Getting the Local Date and Time

23564 The following example uses the `time()` function to calculate the time elapsed, in seconds, since
23565 January 1, 1970 0:00 UTC (the Epoch), `localtime()` to convert that value to a broken-down time,
23566 and `asctime()` to convert the broken-down time values into a printable string.

```
23567 #include <stdio.h>
23568 #include <time.h>
23569
23570     int main(void)
23571     {
23572         time_t result;
23573
23574         result = time(NULL);
23575         printf("%s%ju secs since the Epoch\n",
23576             asctime(localtime(&result)),
23577             (uintmax_t)result);
23578         return(0);
23579     }
```

23580 This example writes the current time to `stdout` in a form like this:

```
23581 Wed Jun 26 10:32:15 1996
23582 835810335 secs since the Epoch
```

23583 Getting the Modification Time for a File

23584 The following example gets the modification time for a file. The `localtime()` function converts the
23585 `time_t` value of the last modification date, obtained by a previous call to `stat()`, into a `tm`
23586 structure that contains the year, month, day, and so on.

```
23587 #include <time.h>
23588 ...
23589 struct stat statbuf;
23590 ...
23591 tm = localtime(&statbuf.st_mtime);
23592 ...
```

23593 Timing an Event

23594 The following example gets the current time, converts it to a string using `localtime()` and
23595 `asctime()`, and prints it to standard output using `fputs()`. It then prints the number of minutes to
23596 an event being timed.

```
23597 #include <time.h>
23598 #include <stdio.h>
23599 ...
23600 time_t now;
23601 ...
23602 time(&now);
23603 printf("The time is ");
23604 fputs(asctime(localtime(&now)), stdout);
23605 printf("There are still %d minutes to the event.\n",
23606
```

23605 minutes_to_event);
23606 ...

23607 APPLICATION USAGE

23608 The *localtime_r()* function is thread-safe and returns values in a user-supplied buffer instead of
23609 possibly using a static data area that may be overwritten by each call.

23610 RATIONALE

23611 None.

23612 FUTURE DIRECTIONS

23613 None.

23614 SEE ALSO

23615 *asctime()*, *clock()*, *ctime()*, *difftime()*, *getdate()*, *gmtime()*, *mktime()*, *strftime()*, *strptime()*, *time()*,
23616 *tzset()*, *utime()*, the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

2

23617 CHANGE HISTORY

23618 First released in Issue 1. Derived from Issue 1 of the SVID.

23619 Issue 5

23620 A note indicating that the *localtime()* function need not be reentrant is added to the
23621 DESCRIPTION.

23622 The *localtime_r()* function is included for alignment with the POSIX Threads Extension.

23623 Issue 6

23624 The *localtime_r()* function is marked as part of the Thread-Safe Functions option.

23625 Extensions beyond the ISO C standard are marked.

23626 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
23627 its avoidance of possibly using a static data area.

23628 The **restrict** keyword is added to the *localtime_r()* prototype for alignment with the
23629 ISO/IEC 9899:1999 standard.

23630 Examples are added.

23631 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/32 is applied, adding the [EOVERFLOW] error. 1
23632

23633 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/55 is applied, updating the error handling 2
23634 for *localtime_r()*. 2

23635 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/56 is applied, adding a requirement that if 2
23636 *localtime_r()* does not set the *tzname* variable, it shall not set the *daylight* or *timezone* variables. On 2
23637 systems supporting XSI, the *daylight*, *timezone*, and *tzname* variables should all be set to provide 2
23638 information for the same timezone. This updates the description of *localtime_r()* to mention 2
23639 *daylight* and *timezone* as well as *tzname*. The SEE ALSO section is updated. 2

23640 NAME

23641 lockf — record locking on files

23642 SYNOPSIS

```
23643 XSI #include <unistd.h>
23644     int lockf(int fildes, int function, off_t size);
23645
```

23646 DESCRIPTION

The *lockf()* function shall lock sections of a file with advisory-mode locks. Calls to *lockf()* from other threads which attempt to lock the locked file section shall either return an error value or block until the section becomes unlocked. All the locks for a process are removed when the process terminates. Record locking with *lockf()* shall be supported for regular files and may be supported for other files.

The *fildes* argument is an open file descriptor. To establish a lock with this function, the file descriptor shall be opened with write-only permission (O_WRONLY) or with read/write permission (O_RDWR).

The *function* argument is a control value which specifies the action to be taken. The permissible values for *function* are defined in <unistd.h> as follows:

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.

23663 F_TEST shall detect if a lock by another process is present on the specified section.

23664 F_LOCK and F_TLOCK shall both lock a section of a file if the section is available.

23665 F_ULOCK shall remove locks from a section of the file.

23666 The *size* argument is the number of contiguous bytes to be locked or unlocked. The section to be locked or unlocked starts at the current offset in the file and extends forward for a positive size or backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is 0, the section from the current offset through the largest possible file offset shall be locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file to be locked because locks may exist past the end-of-file.

23672 The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent locked sections would occur, the sections shall be combined into a single locked section. If the request would cause the number of locks to exceed a system-imposed limit, the request shall fail.

23676 F_LOCK and F_TLOCK requests differ only by the action taken if the section is not available. F_LOCK shall block the calling thread until the section is available. F_TLOCK shall cause the function to fail if the section is already locked by another process.

23679 File locks shall be released on first close by the locking process of any file descriptor for the file.

23680 F_ULOCK requests may release (wholly or in part) one or more locked sections controlled by the process. Locked sections shall be unlocked starting at the current file offset through *size* bytes or to the end-of-file if *size* is (off_t)0. When all of a locked section is not released (that is, when the beginning or end of the area to be unlocked falls within a locked section), the remaining portions of that section shall remain locked by the process. Releasing the center portion of a locked

23685 section shall cause the remaining locked beginning and end portions to become two separate
23686 locked sections. If the request would cause the number of locks in the system to exceed a
23687 system-imposed limit, the request shall fail.

23688 A potential for deadlock occurs if the threads of a process controlling a locked section are
23689 blocked by accessing another process' locked section. If the system detects that deadlock would
23690 occur, *lockf()* shall fail with an [EDEADLK] error.

23691 The interaction between *fcntl()* and *lockf()* locks is unspecified.

23692 Blocking on a section shall be interrupted by any signal.

23693 An F_ULOCK request in which *size* is non-zero and the offset of the last byte of the requested
23694 section is the maximum value for an object of type **off_t**, when the process has an existing lock
23695 in which *size* is 0 and which includes the last byte of the requested section, shall be treated as a
23696 request to unlock from the start of the requested section with a size equal to 0. Otherwise, an
23697 F_ULOCK request shall attempt to unlock only the requested section.

23698 Attempting to lock a section of a file that is associated with a buffered stream produces
23699 unspecified results.

23700 RETURN VALUE

23701 Upon successful completion, *lockf()* shall return 0. Otherwise, it shall return -1, set *errno* to
23702 indicate an error, and existing locks shall not be changed.

23703 ERRORS

23704 The *lockf()* function shall fail if:

23705 [EBADF] The *fildes* argument is not a valid open file descriptor; or *function* is F_LOCK
23706 or F_TLOCK and *fildes* is not a valid file descriptor open for writing.

23707 [EACCES] or [EAGAIN] The *function* argument is F_TLOCK or F_TEST and the section is already
23708 locked by another process.

23710 [EDEADLK] The *function* argument is F_LOCK and a deadlock is detected.

23711 [EINTR] A signal was caught during execution of the function.

23712 [EINVAL] The *function* argument is not one of F_LOCK, F_TLOCK, F_TEST, or
23713 F_ULOCK; or *size* plus the current file offset is less than 0.

23714 [EOVERFLOW] The offset of the first, or if *size* is not 0 then the last, byte in the requested
23715 section cannot be represented correctly in an object of type **off_t**.

23716 The *lockf()* function may fail if:

23717 [EAGAIN] The *function* argument is F_LOCK or F_TLOCK and the file is mapped with
23718 *mmap()*.

23719 [EDEADLK] or [ENOLCK] The *function* argument is F_LOCK, F_TLOCK, or F_ULOCK, and the request
23720 would cause the number of locks to exceed a system-imposed limit.

23722 [EOPNOTSUPP] or [EINVAL] The implementation does not support the locking of files of the type indicated
23723 by the *fildes* argument.

23725 EXAMPLES**23726 Locking a Portion of a File**

23727 In the following example, a file named **/home/cnd/mod1** is being modified. Other processes that
23728 use locking are prevented from changing it during this process. Only the first 10 000 bytes are
23729 locked, and the lock call fails if another process has any part of this area locked already.

```
23730 #include <fcntl.h>
23731 #include <unistd.h>
23732 int fildes;
23733 int status;
23734 ...
23735 fildes = open( "/home/cnd/mod1" , O_RDWR );
23736 status = lockf(fildes, F_TLOCK, (off_t)10000);
```

23737 APPLICATION USAGE

23738 Record-locking should not be used in combination with the *fopen()*, *fread()*, *fwrite()*, and other
23739 *stdio* functions. Instead, the more primitive, non-buffered functions (such as *open()*) should be
23740 used. Unexpected results may occur in processes that do buffering in the user address space. The
23741 process may later read/write data which is/was locked. The *stdio* functions are the most
23742 common source of unexpected buffering.

23743 The *alarm()* function may be used to provide a timeout facility in applications requiring it.

23744 RATIONALE

23745 None.

23746 FUTURE DIRECTIONS

23747 None.

23748 SEE ALSO

23749 *alarm()*, *chmod()*, *close()*, *creat()*, *fcntl()*, *fopen()*, *mmap()*, *open()*, *read()*, *write()*, the Base
23750 Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

23751 CHANGE HISTORY

23752 First released in Issue 4, Version 2.

23753 Issue 5

23754 Moved from X/OPEN UNIX extension to BASE.

23755 Large File Summit extensions are added. In particular, the description of [EINVAL] is clarified
23756 and moved from optional to mandatory status.

23757 A note is added to the DESCRIPTION indicating the effects of attempting to lock a section of a
23758 file that is associated with a buffered stream.

23759 Issue 6

23760 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

23761 NAME

23762 log, logf, logl — natural logarithm function

23763 SYNOPSIS

```
23764 #include <math.h>
23765 double log(double x);
23766 float logf(float x);
23767 long double logl(long double x);
```

23768 DESCRIPTION

23769 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23770 conflict between the requirements described here and the ISO C standard is unintentional. This
23771 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23772 These functions shall compute the natural logarithm of their argument x , $\log_e(x)$.

23773 An application wishing to check for error situations should set *errno* to zero and call
23774 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
23775 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
23776 zero, an error has occurred.

23777 RETURN VALUE

23778 Upon successful completion, these functions shall return the natural logarithm of x .

23779 If x is ± 0 , a pole error shall occur and *log()*, *logf()*, and *logl()* shall return $-\text{HUGE_VAL}$,
23780 $-\text{HUGE_VALF}$, and $-\text{HUGE_VALL}$, respectively.

23781 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
23782 NaN (if supported), or an implementation-defined value shall be returned.

23783 MX If x is NaN, a NaN shall be returned.

23784 If x is 1, $+0$ shall be returned.

23785 If x is $+\text{Inf}$, x shall be returned.

23786 ERRORS

23787 These functions shall fail if:

23788 MX Domain Error The finite value of x is negative, or x is $-\text{Inf}$.

23789 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
23790 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
23791 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
23792 shall be raised.

23793 Pole Error The value of x is zero.

23794 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
23795 then *errno* shall be set to [ERANGE]. If the integer expression
23796 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-
23797 zero floating-point exception shall be raised.

23798 EXAMPLES

23799 None.

23800 APPLICATION USAGE

23801 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
23802 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23803 RATIONALE

23804 None.

23805 FUTURE DIRECTIONS

23806 None.

23807 SEE ALSO

23808 *exp()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *log10()*, *log1p()*, the Base Definitions volume of
23809 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
23810 *<math.h>*

23811 CHANGE HISTORY

23812 First released in Issue 1. Derived from Issue 1 of the SVID.

23813 Issue 5

23814 The DESCRIPTION is updated to indicate how an application should check for an error. This
23815 text was previously published in the APPLICATION USAGE section.

23816 Issue 6

23817 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
23818 The *logf()* and *logl()* functions are added for alignment with the ISO/IEC 9899: 1999 standard.
23819 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
23820 revised to align with the ISO/IEC 9899: 1999 standard.
23821 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are
23822 marked.

23823 NAME

23824 log10, log10f, log10l — base 10 logarithm function

23825 SYNOPSIS

```
23826 #include <math.h>
23827 double log10(double x);
23828 float log10f(float x);
23829 long double log10l(long double x);
```

23830 DESCRIPTION

23831 CX The functionality described on this reference page is aligned with the ISO C standard. Any
23832 conflict between the requirements described here and the ISO C standard is unintentional. This
23833 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

23834 These functions shall compute the base 10 logarithm of their argument x , $\log_{10}(x)$.

23835 An application wishing to check for error situations should set *errno* to zero and call
23836 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
23837 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
23838 zero, an error has occurred.

23839 RETURN VALUE

23840 Upon successful completion, these functions shall return the base 10 logarithm of x .

23841 If x is ± 0 , a pole error shall occur and *log10()*, *log10f()*, and *log10l()* shall return $-\text{HUGE_VAL}$,
23842 $-\text{HUGE_VALF}$, and $-\text{HUGE_VALL}$, respectively.

23843 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
23844 NaN (if supported), or an implementation-defined value shall be returned.

23845 MX If x is NaN, a NaN shall be returned.

23846 If x is 1, +0 shall be returned.

23847 If x is $+\text{Inf}$, $+\text{Inf}$ shall be returned.

23848 ERRORS

23849 These functions shall fail if:

23850 MX Domain Error The finite value of x is negative, or x is $-\text{Inf}$.

23851 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
23852 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
23853 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
23854 shall be raised.

23855 Pole Error The value of x is zero.

23856 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
23857 then *errno* shall be set to [ERANGE]. If the integer expression
23858 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-
23859 zero floating-point exception shall be raised.

23860 EXAMPLES

23861 None.

23862 APPLICATION USAGE

23863 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
23864 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23865 RATIONALE

23866 None.

23867 FUTURE DIRECTIONS

23868 None.

23869 SEE ALSO

23870 *feclearexcept()*, *fetestexcept()*, *isnan()*, *log()*, *pow()*, the Base Definitions volume of
23871 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
23872 *<math.h>*

23873 CHANGE HISTORY

23874 First released in Issue 1. Derived from Issue 1 of the SVID.

23875 Issue 5

23876 The DESCRIPTION is updated to indicate how an application should check for an error. This
23877 text was previously published in the APPLICATION USAGE section.

23878 Issue 6

23879 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

23880 The *log10f()* and *log10l()* functions are added for alignment with the ISO/IEC 9899:1999
23881 standard.

23882 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
23883 revised to align with the ISO/IEC 9899:1999 standard.

23884 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
23885 marked.

23886 NAME

23887 log1p, log1pf, log1pl — compute a natural logarithm

23888 SYNOPSIS

```
23889 #include <math.h>
23890 double log1p(double x);
23891 float log1pf(float x);
23892 long double log1pl(long double x);
```

23893 DESCRIPTION

23894 CX 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.

23897 These functions shall compute $\log_e(1.0 + x)$.

23898 An application wishing to check for error situations should set *errno* to zero and call
 23899 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
 23900 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
 23901 zero, an error has occurred.

23902 RETURN VALUE

23903 Upon successful completion, these functions shall return the natural logarithm of $1.0 + x$.

23904 If x is -1 , a pole error shall occur and *log1p()*, *log1pf()*, and *log1pl()* shall return $-\text{HUGE_VAL}$,
 23905 $-\text{HUGE_VALF}$, and $-\text{HUGE_VALL}$, respectively.

23906 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
 23907 NaN (if supported), or an implementation-defined value shall be returned.

23908 MX If x is NaN, a NaN shall be returned.

23909 If x is ± 0 , or $+\text{Inf}$, x shall be returned.

23910 If x is subnormal, a range error may occur and x should be returned.

23911 ERRORS

23912 These functions shall fail if:

23913 MX Domain Error The finite value of x is less than -1 , or x is $-\text{Inf}$.

23914 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 23915 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
 23916 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
 23917 shall be raised.

23918 Pole Error The value of x is -1 .

23919 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 23920 then *errno* shall be set to [ERANGE]. If the integer expression
 23921 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by-
 23922 zero floating-point exception shall be raised.

23923 These functions may fail if:

23924 MX Range Error The value of x is subnormal.

23925 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 23926 then *errno* shall be set to [ERANGE]. If the integer expression
 23927 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 23928 floating-point exception shall be raised.

23929 EXAMPLES

23930 None.

23931 APPLICATION USAGE

23932 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
23933 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23934 RATIONALE

23935 None.

23936 FUTURE DIRECTIONS

23937 None.

23938 SEE ALSO

23939 *feclearexcept()*, *fetestexcept()*, *log()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section
23940 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

23941 CHANGE HISTORY

23942 First released in Issue 4, Version 2.

23943 Issue 5

23944 Moved from X/OPEN UNIX extension to BASE.

23945 Issue 6

23946 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

23947 The *log1p()* function is no longer marked as an extension.

23948 The *log1pf()* and *log1pl()* functions are added for alignment with the ISO/IEC 9899:1999
23949 standard.

23950 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
23951 revised to align with the ISO/IEC 9899:1999 standard.

23952 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
23953 marked.

23954 NAME

23955 `log2, log2f, log2l — compute base 2 logarithm functions`

23956 SYNOPSIS

```
23957     #include <math.h>
23958
23959     double log2(double x);
23960     float log2f(float x);
23961     long double log2l(long double x);
```

23961 DESCRIPTION

23962 CX 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.

23965 These functions shall compute the base 2 logarithm of their argument x , $\log_2(x)$.

23966 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.

23970 RETURN VALUE

23971 Upon successful completion, these functions shall return the base 2 logarithm of x .

23972 If x is ± 0 , a pole error shall occur and *log2()*, *log2f()*, and *log2l()* shall return $-HUGE_VAL$, $-HUGE_VALF$, and $-HUGE_VALL$, respectively.

23974 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 NaN (if supported), or an implementation-defined value shall be returned.

23976 MX If x is NaN, a NaN shall be returned.

23977 If x is 1, $+0$ shall be returned.

23978 If x is $+Inf$, x shall be returned.

23979 ERRORS

23980 These functions shall fail if:

23981 MX Domain Error The finite value of x is less than zero, or x is $-\text{Inf}$.

23982 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero, 23983 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling* 23984 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception 23985 shall be raised.

23986 Pole Error The value of x is zero.

23987 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero, 23988 then *errno* shall be set to [ERANGE]. If the integer expression 23989 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by- 23990 zero floating-point exception shall be raised.

23991 EXAMPLES

23992 None.

23993 APPLICATION USAGE

23994 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
23995 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

23996 RATIONALE

23997 None.

23998 FUTURE DIRECTIONS

23999 None.

24000 SEE ALSO

24001 *feclearexcept()*, *fetestexcept()*, *log()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section
24002 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

24003 CHANGE HISTORY

24004 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

24005 NAME

24006 **logb, logbf, logbl** — radix-independent exponent

24007 SYNOPSIS

```
24008     #include <math.h>
24009
24010     double logb(double x);
24011     float logbf(float x);
24012     long double logbl(long double x);
```

24012 DESCRIPTION

24013 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24014 conflict between the requirements described here and the ISO C standard is unintentional. This
24015 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24016 These functions shall compute the exponent of x , which is the integral part of $\log_r |x|$, as a
24017 signed floating-point value, for non-zero x , where r is the radix of the machine's floating-point
24018 arithmetic, which is the value of `FLT_RADIX` defined in the `<float.h>` header.

24019 If x is subnormal it is treated as though it were normalized; thus for finite positive x :

```
24020     1 <= x * FLT_RADIX-logb(x) < FLT_RADIX
```

24021 An application wishing to check for error situations should set `errno` to zero and call
24022 `feclearexcept(FE_ALL_EXCEPT)` before calling these functions. On return, if `errno` is non-zero or
24023 `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)` is non-
24024 zero, an error has occurred.

24025 RETURN VALUE

24026 Upon successful completion, these functions shall return the exponent of x .

24027 If x is ± 0 , a pole error shall occur and `logb()`, `logbf()`, and `logbl()` shall return `-HUGE_VAL`,
24028 `-HUGE_VALF`, and `-HUGE_VALL`, respectively.

24029 MX If x is `NaN`, a `NaN` shall be returned.

24030 If x is `+Inf`, `+Inf` shall be returned.

24031 ERRORS

24032 These functions shall fail if:

24033 Pole Error The value of x is ± 0 .

24034 If the integer expression (`math_errhandling & MATH_ERRNO`) is non-zero,
24035 then `errno` shall be set to `[ERANGE]`. If the integer expression
24036 (`math_errhandling & MATH_ERREXCEPT`) is non-zero, then the divide-by-
24037 zero floating-point exception shall be raised.

24038 EXAMPLES

24039 None.

24040 APPLICATION USAGE

24041 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling &`
24042 `MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

24043 RATIONALE

24044 None.

24045 FUTURE DIRECTIONS

24046 None.

24047 SEE ALSO

24048 *feclearexcept()*, *fetestexcept()*, *ilogb()*, *scalb()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <float.h>, <math.h>

24050 CHANGE HISTORY

24051 First released in Issue 4, Version 2.

24052 Issue 5

24053 Moved from X/OPEN UNIX extension to BASE.

24054 Issue 6

24055 The *logb()* function is no longer marked as an extension.

24056 The *logbf()* and *logbl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

24057 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

24059 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

24060

24061 **NAME**24062 **logf, logl** — natural logarithm function24063 **SYNOPSIS**24064

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

24065

```
float logf(float x);
```

24066

```
long double logl(long double x);
```

24067 **DESCRIPTION**24068 Refer to *log()*.

24069 NAME

24070 longjmp — non-local goto

24071 SYNOPSIS

24072 #include <setjmp.h>
24073 void longjmp(jmp_buf env, int val);

24074 DESCRIPTION

24075 CX 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.

24078 The *longjmp()* function shall restore the environment saved by the most recent invocation of *setjmp()* in the same thread, with the corresponding **jmp_buf** argument. If there is no such invocation, or if the function containing the invocation of *setjmp()* has terminated execution in the interim, or if the invocation of *setjmp()* was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined. It is unspecified whether *longjmp()* restores the signal mask, leaves the signal mask unchanged, or restores it to its value at the time *setjmp()* was called.

24085 All accessible objects have values, and all other components of the abstract machine have state (for example, floating-point status flags and open files), as of the time *longjmp()* was called, except that the values of objects of automatic storage duration are unspecified if they meet all the following conditions:

- They are local to the function containing the corresponding *setjmp()* invocation.
- They do not have volatile-qualified type.
- They are changed between the *setjmp()* invocation and *longjmp()* call.

24092 CX As it bypasses the usual function call and return mechanisms, *longjmp()* shall execute correctly in contexts of interrupts, signals, and any of their associated functions. However, if *longjmp()* is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

24096 The effect of a call to *longjmp()* where initialization of the **jmp_buf** structure was not performed in the calling thread is undefined.

24098 RETURN VALUE

24099 After *longjmp()* is completed, program execution continues as if the corresponding invocation of *setjmp()* had just returned the value specified by *val*. The *longjmp()* function shall not cause *setjmp()* to return 0; if *val* is 0, *setjmp()* shall return 1.

24102 ERRORS

24103 No errors are defined.

24104 EXAMPLES

24105 None.

24106 APPLICATION USAGE

24107 Applications whose behavior depends on the value of the signal mask should not use *longjmp()* and *setjmp()*, since their effect on the signal mask is unspecified, but should instead use the *siglongjmp()* and *sigsetjmp()* functions (which can save and restore the signal mask under application control).

24111 RATIONALE

24112 None.

24113 FUTURE DIRECTIONS

24114 None.

24115 SEE ALSO

24116 *setjmp()*, *sigaction()*, *siglongjmp()*, *sigsetjmp()*, the Base Definitions volume of
24117 IEEE Std 1003.1-2001, <setjmp.h>

24118 CHANGE HISTORY

24119 First released in Issue 1. Derived from Issue 1 of the SVID.

24120 Issue 5

24121 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

24122 Issue 6

24123 Extensions beyond the ISO C standard are marked.

24124 The following new requirements on POSIX implementations derive from alignment with the
24125 Single UNIX Specification:

- The DESCRIPTION now explicitly makes *longjmp()*'s effect on the signal mask unspecified.

24127 The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

24128 **NAME**

24129 lrand48 — generate uniformly distributed pseudo-random non-negative long integers

24130 **SYNOPSIS**

24131 XSI #include <stdlib.h>

24132 long lrand48(void);

24133

24134 **DESCRIPTION**

24135 Refer to *drand48()*.

24136 NAME

24137 lrint, lrintf, lrintl — round to nearest integer value using current rounding direction

24138 SYNOPSIS

```
24139 #include <math.h>
24140 long lrint(double x);
24141 long lrintf(float x);
24142 long lrintl(long double x);
```

24143 DESCRIPTION

24144 CX 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.

24147 These functions shall round their argument to the nearest integer value, rounding according to
24148 the current rounding direction.

24149 An application wishing to check for error situations should set *errno* to zero and call
24150 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
24151 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
24152 zero, an error has occurred.

24153 RETURN VALUE

24154 Upon successful completion, these functions shall return the rounded integer value.

24155 MX If *x* is NaN, a domain error shall occur and an unspecified value is returned.

24156 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

24157 If *x* is -Inf, a domain error shall occur and an unspecified value is returned.

24158 If the correct value is positive and too large to represent as a **long**, a domain error shall occur
24159 and an unspecified value is returned.

24160 If the correct value is negative and too large to represent as a **long**, a domain error shall occur
24161 and an unspecified value is returned.

24162 ERRORS

24163 These functions shall fail if:

24164 MX Domain Error The *x* argument is NaN or ±Inf, or the correct value is not representable as an
24165 integer.

24166 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
24167 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
24168 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
24169 shall be raised.

24170 EXAMPLES

24171 None.

24172 APPLICATION USAGE

24173 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
24174 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

24175 RATIONALE

24176 These functions provide floating-to-integer conversions. They round according to the current
24177 rounding direction. If the rounded value is outside the range of the return type, the numeric
24178 result is unspecified and the invalid floating-point exception is raised. When they raise no other
24179 floating-point exception and the result differs from the argument, they raise the inexact

24180 floating-point exception.

24181 **FUTURE DIRECTIONS**

24182 None.

24183 **SEE ALSO**

24184 *feclearexcept()*, *fetestexcept()*, *lrint()*, the Base Definitions volume of IEEE Std 1003.1-2001,
24185 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

24186 **CHANGE HISTORY**

24187 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

24188 NAME

24189 lround, lroundf, lroundl — round to nearest integer value

24190 SYNOPSIS

```
24191 #include <math.h>
24192 long lround(double x);
24193 long lroundf(float x);
24194 long lroundl(long double x);
```

24195 DESCRIPTION

24196 CX 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.

24199 These functions shall round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction.

24201 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.

24205 RETURN VALUE

24206 Upon successful completion, these functions shall return the rounded integer value.

24207 MX If *x* is NaN, a domain error shall occur and an unspecified value is returned.

24208 If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

24209 If *x* is -Inf, a domain error shall occur and an unspecified value is returned.

24210 If the correct value is positive and too large to represent as a **long**, a domain error shall occur and an unspecified value is returned.

24212 If the correct value is negative and too large to represent as a **long**, a domain error shall occur and an unspecified value is returned.

24214 ERRORS

24215 These functions shall fail if:

24216 MX Domain Error The *x* argument is NaN or ±Inf, or the correct value is not representable as an integer.

24218 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.

24222 EXAMPLES

24223 None.

24224 APPLICATION USAGE

24225 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

24227 RATIONALE

24228 These functions differ from the *lrint()* functions in the default rounding direction, with the 24229 *lround()* functions rounding halfway cases away from zero and needing not to raise the inexact 24230 floating-point exception for non-integer arguments that round to within the range of the return 24231 type.

24232 FUTURE DIRECTIONS

24233 None.

24234 SEE ALSO

24235 *feclearexcept()*, *fetestexcept()*, *llround()*, the Base Definitions volume of IEEE Std 1003.1-2001,
24236 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

24237 CHANGE HISTORY

24238 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

24239 NAME

24240 lsearch, lfind — linear search and update

24241 SYNOPSIS

```
24242 XSI #include <search.h>
24243 void *lsearch(const void *key, void *base, size_t *nelp, size_t width,
24244     int (*compar)(const void *, const void *));
24245 void *lfind(const void *key, const void *base, size_t *nelp,
24246     size_t width, int (*compar)(const void *, const void *));
24247
```

24248 DESCRIPTION

24249 The *lsearch()* function shall linearly search the table and return a pointer into the table for the
 24250 matching entry. If the entry does not occur, it shall be added at the end of the table. The *key*
 24251 argument points to the entry to be sought in the table. The *base* argument points to the first
 24252 element in the table. The *width* argument is the size of an element in bytes. The *nelp* argument
 24253 points to an integer containing the current number of elements in the table. The integer to which
 24254 *nelp* points shall be incremented if the entry is added to the table. The *compar* argument points to
 24255 a comparison function which the application shall supply (for example, *strcmp()*). It is called
 24256 with two arguments that point to the elements being compared. The application shall ensure
 24257 that the function returns 0 if the elements are equal, and non-zero otherwise.

24258 The *lfind()* function shall be equivalent to *lsearch()*, except that if the entry is not found, it is not
 24259 added to the table. Instead, a null pointer is returned.

24260 RETURN VALUE

24261 If the searched for entry is found, both *lsearch()* and *lfind()* shall return a pointer to it. Otherwise,
 24262 *lfind()* shall return a null pointer and *lsearch()* shall return a pointer to the newly added element.

24263 Both functions shall return a null pointer in case of error.

24264 ERRORS

24265 No errors are defined.

24266 EXAMPLES

24267 Storing Strings in a Table

24268 This fragment reads in less than or equal to TABSIZE strings of length less than or equal to
 24269 ELSIZE and stores them in a table, eliminating duplicates.

```
24270 #include <stdio.h>
24271 #include <string.h>
24272 #include <search.h>
24273
24274 #define TABSIZE 50
24275 #define ELSIZE 120
24276
24277     ...
24278     char line[ELSIZE], tab[TABSIZE][ELSIZE];
24279     size_t nel = 0;
24280
24281     ...
24282     while (fgets(line, ELSIZE, stdin) != NULL && nel < TABSIZE)
24283         (void) lsearch(line, tab, &nel,
24284             ELSIZE, (int (*)(const void *, const void *)) strcmp);
24285
24286     ...
```

24283 **Finding a Matching Entry**

24284 The following example finds any line that reads "This is a test.".

```
24285 #include <search.h>
24286 #include <string.h>
24287 ...
24288 char line[ELSIZE], tab[TABSIZE][ELSIZE];
24289 size_t nel = 0;
24290 char *findline;
24291 void *entry;
24292
24293 findline = "This is a test.\n";
24294 entry = lfind(findline, tab, &nel, ELSIZE,
24295   int (*)(const void *, const void *)) strcmp);
```

24295 **APPLICATION USAGE**

24296 The comparison function need not compare every byte, so arbitrary data may be contained in
24297 the elements in addition to the values being compared.

24298 Undefined results can occur if there is not enough room in the table to add a new item.

24299 **RATIONALE**

24300 None.

24301 **FUTURE DIRECTIONS**

24302 None.

24303 **SEE ALSO**

24304 *hcreate()*, *tsearch()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**search.h**>

24305 **CHANGE HISTORY**

24306 First released in Issue 1. Derived from Issue 1 of the SVID.

24307 **Issue 6**

24308 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

24309 NAME

24310 lseek — move the read/write file offset

24311 SYNOPSIS

```
24312 #include <unistd.h>
24313 off_t lseek(int fildes, off_t offset, int whence);
```

24314 DESCRIPTION

24315 The *lseek()* function shall set the file offset for the open file description associated with the file descriptor *fildes*, as follows:

- If *whence* is SEEK_SET, the file offset shall be set to *offset* bytes.
- If *whence* is SEEK_CUR, the file offset shall be set to its current location plus *offset*.
- If *whence* is SEEK_END, the file offset shall be set to the size of the file plus *offset*.

24320 The symbolic constants SEEK_SET, SEEK_CUR, and SEEK_END are defined in <unistd.h>.

24321 The behavior of *lseek()* on devices which are incapable of seeking is implementation-defined.
24322 The value of the file offset associated with such a device is undefined.

24323 The *lseek()* function shall allow the file offset to be set beyond the end of the existing data in the
24324 file. If data is later written at this point, subsequent reads of data in the gap shall return bytes
24325 with the value 0 until data is actually written into the gap.

24326 The *lseek()* function shall not, by itself, extend the size of a file.

24327 SHM If *fildes* refers to a shared memory object, the result of the *lseek()* function is unspecified.

24328 TYM If *fildes* refers to a typed memory object, the result of the *lseek()* function is unspecified.

24329 RETURN VALUE

24330 Upon successful completion, the resulting offset, as measured in bytes from the beginning of the
24331 file, shall be returned. Otherwise, (off_t)-1 shall be returned, *errno* shall be set to indicate the
24332 error, and the file offset shall remain unchanged.

24333 ERRORS

24334 The *lseek()* function shall fail if:

24335 [EBADF]	The <i>fildes</i> argument is not an open file descriptor.
24336 [EINVAL]	The <i>whence</i> argument is not a proper value, or the resulting file offset would 24337 be negative for a regular file, block special file, or directory.
24338 [EOVERFLOW]	The resulting file offset would be a value which cannot be represented 24339 correctly in an object of type off_t.
24340 [ESPIPE]	The <i>fildes</i> argument is associated with a pipe, FIFO, or socket.

24341 EXAMPLES

24342 None.

24343 APPLICATION USAGE

24344 None.

24345 RATIONALE

24346 The ISO C standard includes the functions *fgetpos()* and *fsetpos()*, which work on very large files
24347 by use of a special positioning type.

24348 Although *lseek()* may position the file offset beyond the end of the file, this function does not
24349 itself extend the size of the file. While the only function in IEEE Std 1003.1-2001 that may directly

24350 extend the size of the file is *write()*, *truncate()*, and *ftruncate()*, several functions originally
24351 derived from the ISO C standard, such as *fwrite()*, *fprintf()*, and so on, may do so (by causing
24352 calls on *write()*).

24353 An invalid file offset that would cause [EINVAL] to be returned may be both implementation-
24354 defined and device-dependent (for example, memory may have few invalid values). A negative
24355 file offset may be valid for some devices in some implementations.

24356 The POSIX.1-1990 standard did not specifically prohibit *lseek()* from returning a negative offset.
24357 Therefore, an application was required to clear *errno* prior to the call and check *errno* upon return
24358 to determine whether a return value of (*off_t*)−1 is a negative offset or an indication of an error
24359 condition. The standard developers did not wish to require this action on the part of a
24360 conforming application, and chose to require that *errno* be set to [EINVAL] when the resulting
24361 file offset would be negative for a regular file, block special file, or directory.

24362 FUTURE DIRECTIONS

24363 None.

24364 SEE ALSO

24365 *open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

24366 CHANGE HISTORY

24367 First released in Issue 1. Derived from Issue 1 of the SVID.

24368 Issue 5

24369 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

24370 Large File Summit extensions are added.

24371 Issue 6

24372 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

24373 The following new requirements on POSIX implementations derive from alignment with the
24374 Single UNIX Specification:

- The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
24376 required for conforming implementations of previous POSIX specifications, it was not
24377 required for UNIX applications.
- The [EOVERFLOW] error condition is added. This change is to support large files.

24379 An additional [ESPIPE] error condition is added for sockets.

24380 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
24381 *lseek()* results are unspecified for typed memory objects.

24382 NAME

24383 lstat — get symbolic link status

24384 SYNOPSIS

24385 #include <sys/stat.h>
24386 int lstat(const char *restrict path, struct stat *restrict buf);

24387 DESCRIPTION

24388 The *lstat()* function shall be equivalent to *stat()*, except when *path* refers to a symbolic link. In
24389 that case *lstat()* shall return information about the link, while *stat()* shall return information
24390 about the file the link references.

24391 For symbolic links, the *st_mode* member shall contain meaningful information when used with
24392 the file type macros, and the *st_size* member shall contain the length of the pathname contained
24393 in the symbolic link. File mode bits and the contents of the remaining members of the **stat**
24394 structure are unspecified. The value returned in the *st_size* member is the length of the contents
24395 of the symbolic link, and does not count any trailing null.

24396 RETURN VALUE

24397 Upon successful completion, *lstat()* shall return 0. Otherwise, it shall return -1 and set *errno* to
24398 indicate the error.

24399 ERRORS

24400 The *lstat()* function shall fail if:

- 24401 [EACCES] A component of the path prefix denies search permission.
- 24402 [EIO] An error occurred while reading from the file system.
- 24403 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.
- 24405 [ENAMETOOLONG] The length of a pathname exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
- 24408 [ENOTDIR] A component of the path prefix is not a directory.
- 24409 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.
- 24410 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

24413 The *lstat()* function may fail if:

- 24414 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
- 24416 [ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.
- 24419 [EOVERFLOW] One of the members is too large to store into the structure pointed to by the *buf* argument.

24421 EXAMPLES

24422 Obtaining Symbolic Link Status Information

24423 The following example shows how to obtain status information for a symbolic link named
24424 **/modules/pass1**. The structure variable *buffer* is defined for the **stat** structure. If the *path*
24425 argument specified the filename for the file pointed to by the symbolic link (**/home/cnd/mod1**),
24426 the results of calling the function would be the same as those returned by a call to the **stat()**
24427 function.

```
24428      #include <sys/stat.h>
24429
24430      struct stat buffer;
24431      int status;
24432      ...
24433      status = lstat("/modules/pass1", &buffer);
```

24433 APPLICATION USAGE

24434 None.

24435 RATIONALE

24436 The *lstat()* function is not required to update the time-related fields if the named file is not a
24437 symbolic link. While the *st_uid*, *st_gid*, *st_atime*, *st_mtime*, and *st_ctime* members of the **stat**
24438 structure may apply to a symbolic link, they are not required to do so. No functions in
24439 IEEE Std 1003.1-2001 are required to maintain any of these time fields.

24440 FUTURE DIRECTIONS

24441 None.

24442 SEE ALSO

24443 *fstat()*, *readlink()*, *stat()*, *symlink()*, the Base Definitions volume of IEEE Std 1003.1-2001,
24444 <sys/stat.h>

24445 CHANGE HISTORY

24446 First released in Issue 4, Version 2.

24447 Issue 5

24448 Moved from X/OPEN UNIX extension to BASE.

24449 Large File Summit extensions are added.

24450 Issue 6

24451 The following changes were made to align with the IEEE P1003.1a draft standard:

- 24452 • This function is now mandatory.
- 24453 • The [ELOOP] optional error condition is added.

24454 The **restrict** keyword is added to the *lstat()* prototype for alignment with the ISO/IEC 9899:1999
24455 standard.

24456 NAME

24457 makecontext, swapcontext — manipulate user contexts

24458 SYNOPSIS

24459 OB XSI	#include <ucontext.h>	2
	void makecontext(ucontext_t *ucp, void (*func)(),	2
	int argc, ...);	2
	int swapcontext(ucontext_t *restrict oucp,	
	const ucontext_t *restrict ucp);	
24464		

24465 DESCRIPTION

24466 The *makecontext()* function shall modify the context specified by *ucp*, which has been initialized
 24467 using *getcontext()*. When this context is resumed using *swapcontext()* or *setcontext()*, program
 24468 execution shall continue by calling *func*, passing it the arguments that follow *argc* in the
 24469 *makecontext()* call.

24470 Before a call is made to *makecontext()*, the application shall ensure that the context being
 24471 modified has a stack allocated for it. The application shall ensure that the value of *argc* matches
 24472 the number of arguments of type **int** passed to *func*; otherwise, the behavior is undefined. 1

24473 The *uc_link* member is used to determine the context that shall be resumed when the context
 24474 being modified by *makecontext()* returns. The application shall ensure that the *uc_link* member is
 24475 initialized prior to the call to *makecontext()*.

24476 The *swapcontext()* function shall save the current context in the context structure pointed to by
 24477 *oucp* and shall set the context to the context structure pointed to by *ucp*.

24478 RETURN VALUE

24479 Upon successful completion, *swapcontext()* shall return 0. Otherwise, -1 shall be returned and
 24480 *errno* set to indicate the error.

24481 ERRORS

24482 The *swapcontext()* function shall fail if:

24483 [ENOMEM] The *ucp* argument does not have enough stack left to complete the operation.

24484 EXAMPLES

24485 The following example illustrates the use of *makecontext()*:

```
24486 #include <stdio.h>
24487 #include <ucontext.h>
24488
24489 static ucontext_t ctx[3];
24490
24491 static void
24492 f1 (void)
24493 {
24494     puts("start f1");
24495     swapcontext(&ctx[1], &ctx[2]);
24496     puts("finish f1");
24497 }
24498
24499 static void
24500 f2 (void)
24501 {
24502     puts("start f2");
24503     swapcontext(&ctx[2], &ctx[1]);
```

```

24501         puts("finish f2");
24502     }
24503     int
24504     main (void)
24505     {
24506         char st1[8192];
24507         char st2[8192];
24508         getcontext(&ctx[1]);
24509         ctx[1].uc_stack.ss_sp = st1;
24510         ctx[1].uc_stack.ss_size = sizeof st1;
24511         ctx[1].uc_link = &ctx[0];
24512         makecontext(&ctx[1], f1, 0);
24513         getcontext(&ctx[2]);
24514         ctx[2].uc_stack.ss_sp = st2;
24515         ctx[2].uc_stack.ss_size = sizeof st2;
24516         ctx[2].uc_link = &ctx[1];
24517         makecontext(&ctx[2], f2, 0);
24518         swapcontext(&ctx[0], &ctx[2]);
24519         return 0;
24520     }

```

24521 APPLICATION USAGE

24522 The obsolescent functions *getcontext()*, *makecontext()*, and *swapcontext()* can be replaced using 2
 24523 POSIX threads functions. 2

24524 RATIONALE

24525 With the incorporation of the ISO/IEC 9899:1999 standard into this specification it was found 2
 24526 that the ISO C standard (Subclause 6.11.6) specifies that the use of function declarators with 2
 24527 empty parentheses is an obsolescent feature. Therefore, using the function prototype: 2

24528 `void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);` 2

24529 is making use of an obsolescent feature of the ISO C standard. Therefore, a strictly conforming 2
 24530 POSIX application cannot use this form. Therefore, use of *getcontext()*, *makecontext()*, and 2
 24531 *swapcontext()* is marked obsolescent. 2

24532 There is no way in the ISO C standard to specify a non-obsolescent function prototype 2
 24533 indicating that a function will be called with an arbitrary number (including zero) of arguments 2
 24534 of arbitrary types (including integers, pointers to data, pointers to functions, and composite 2
 24535 types). 2

24536 Replacing *makecontext()* with a number of ISO C standard-compatible functions handing various 2
 24537 numbers and types of arguments would have forced all existing uses of *makecontext()* to be 2
 24538 rewritten for little or no gain. There are very few applications today that use the **context()* 2
 24539 routines. Those that do use them are almost always using them to implement co-routines. By 2
 24540 maintaining the XSH, Issue 5 specification for *makecontext()*, existing applications will continue 2
 24541 to work, although they won't be able to be classified as strictly conforming applications. 2

24542 There is no way in the ISO C standard (without using obsolescent behavior) to specify 2
 24543 functionality that was standard, strictly conforming behavior in the XSH, Issue 5 specification 2
 24544 using the ISO C standard. Threads can be used to implement the functionality provided by 2
 24545 *makecontext()*, *getcontext()*, and *swapcontext()* but they are more complex to use. It was felt 2
 24546 inventing new ISO C standard-compatible interfaces that describe what can be done with the 2
 24547 XSH, Issue 5 functions and then converting applications to use them would cause more 2

24548	difficulty than just converting applications that use them to use threads instead.	2
24549	FUTURE DIRECTIONS	
24550	None.	
24551	SEE ALSO	
24552	<i>exit()</i> , <i>getcontext()</i> , <i>sigaction()</i> , <i>sigprocmask()</i> , the Base Definitions volume of	
24553	IEEE Std 1003.1-2001, <ucontext.h>	
24554	CHANGE HISTORY	
24555	First released in Issue 4, Version 2.	
24556	Issue 5	
24557	Moved from X/OPEN UNIX extension to BASE.	
24558	In the ERRORS section, the description of [ENOMEM] is changed to apply to <i>swapcontext()</i> only.	
24559	Issue 6	
24560	The DESCRIPTION is updated to avoid use of the term “must” for application requirements.	
24561	The restrict keyword is added to the <i>swapcontext()</i> prototype for alignment with the	
24562	ISO/IEC 9899:1999 standard.	
24563	IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/33 is applied, clarifying that the	1
24564	arguments passed to <i>func</i> are of type int .	1
24565	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/57 is applied, obsoleting the functions	2
24566	and giving advice on the alternatives. Changes are made to the SYNOPSIS, APPLICATION	2
24567	USAGE, and RATIONALE sections.	2

24568 NAME

24569 malloc — a memory allocator

24570 SYNOPSIS

```
24571        #include <stdlib.h>
24572        void *malloc(size_t size);
```

24573 DESCRIPTION

24574 CX 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.

24577 The *malloc()* function shall allocate unused space for an object whose size in bytes is specified by *size* and whose value is unspecified.

24579 The order and contiguity of storage allocated by successive calls to *malloc()* is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-defined: the value returned shall be either a null pointer or a unique pointer.

24587 RETURN VALUE

24588 Upon successful completion with *size* not equal to 0, *malloc()* shall return a pointer to the allocated space. If *size* is 0, either a null pointer or a unique pointer that can be successfully passed to *free()* shall be returned. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

24592 ERRORS

24593 The *malloc()* function shall fail if:

24594 CX [ENOMEM] Insufficient storage space is available.

24595 EXAMPLES

24596 None.

24597 APPLICATION USAGE

24598 None.

24599 RATIONALE

24600 None.

24601 FUTURE DIRECTIONS

24602 None.

24603 SEE ALSO

24604 *calloc()*, *free()*, *realloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

24605 CHANGE HISTORY

24606 First released in Issue 1. Derived from Issue 1 of the SVID.

24607 Issue 6

24608 Extensions beyond the ISO C standard are marked.

24609 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

24611
24612

- In the RETURN VALUE section, the requirement to set *errno* to indicate an error is added.
- The [ENOMEM] error condition is added.

24613 NAME

24614 mblen — get number of bytes in a character

24615 SYNOPSIS

```
24616 #include <stdlib.h>
24617 int mblen(const char *s, size_t n);
```

24618 DESCRIPTION

24619 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24620 conflict between the requirements described here and the ISO C standard is unintentional. This
24621 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24622 If *s* is not a null pointer, *mblen()* shall determine the number of bytes constituting the character
24623 pointed to by *s*. Except that the shift state of *mbtowc()* is not affected, it shall be equivalent to:

```
24624 mbtowc((wchar_t *)0, s, n);
```

24625 The implementation shall behave as if no function defined in this volume of
24626 IEEE Std 1003.1-2001 calls *mblen()*.

24627 The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a
24628 state-dependent encoding, this function shall be placed into its initial state by a call for which its
24629 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null
24630 pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a
24631 null pointer shall cause this function to return a non-zero value if encodings have state
24632 dependency, and 0 otherwise. If the implementation employs special bytes to change the shift
24633 state, these bytes shall not produce separate wide-character codes, but shall be grouped with an
24634 adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be
24635 unspecified.

24636 RETURN VALUE

24637 If *s* is a null pointer, *mblen()* shall return a non-zero or 0 value, if character encodings,
24638 respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mblen()* shall
24639 either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the
24640 character (if the next *n* or fewer bytes form a valid character), or return -1 (if they do not form a
24641 CX valid character) and may set *errno* to indicate the error. In no case shall the value returned be
24642 greater than *n* or the value of the {MB_CUR_MAX} macro.

24643 ERRORS

24644 The *mblen()* function may fail if:

24645 XSI [EILSEQ] Invalid character sequence is detected.

24646 EXAMPLES

24647 None.

24648 APPLICATION USAGE

24649 None.

24650 RATIONALE

24651 None.

24652 FUTURE DIRECTIONS

24653 None.

24654 **SEE ALSO**

24655 *mbtowc()*, *mbstowcs()*, *wctomb()*, *wcstombs()*, the Base Definitions volume of
24656 IEEE Std 1003.1-2001, <stdlib.h>

24657 **CHANGE HISTORY**

24658 First released in Issue 4. Aligned with the ISO C standard.

24659 NAME

24660 mbrlen — get number of bytes in a character (restartable)

24661 SYNOPSIS

```
24662     #include <wchar.h>
24663
24664     size_t mbrlen(const char *restrict s, size_t n,
24665                   mbstate_t *restrict ps);
```

24665 DESCRIPTION

24666 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24667 conflict between the requirements described here and the ISO C standard is unintentional. This
24668 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24669 If *s* is not a null pointer, *mbrlen()* shall determine the number of bytes constituting the character
24670 pointed to by *s*. It shall be equivalent to:

```
24671     mbstate_t internal;
24672     mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);
```

24673 If *ps* is a null pointer, the *mbrlen()* function shall use its own internal **mbstate_t** object, which is
24674 initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object
24675 pointed to by *ps* shall be used to completely describe the current conversion state of the
24676 associated character sequence. The implementation shall behave as if no function defined in this
24677 volume of IEEE Std 1003.1-2001 calls *mbrlen()*.

24678 The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

24679 RETURN VALUE

24680 The *mbrlen()* function shall return the first of the following that applies:

24681 0 If the next *n* or fewer bytes complete the character that corresponds to the null
24682 wide character.

24683 *positive* If the next *n* or fewer bytes complete a valid character; the value returned shall
24684 be the number of bytes that complete the character.

24685 (*size_t*)−2 If the next *n* bytes contribute to an incomplete but potentially valid character,
24686 and all *n* bytes have been processed. When *n* has at least the value of the
24687 {MB_CUR_MAX} macro, this case can only occur if *s* points at a sequence of
24688 redundant shift sequences (for implementations with state-dependent
24689 encodings).

24690 (*size_t*)−1 If an encoding error occurs, in which case the next *n* or fewer bytes do not
24691 contribute to a complete and valid character. In this case, [EILSEQ] shall be
24692 stored in *errno* and the conversion state is undefined.

24693 ERRORS

24694 The *mbrlen()* function may fail if:

24695 [EINVAL] *ps* points to an object that contains an invalid conversion state.

24696 [EILSEQ] Invalid character sequence is detected.

24697 EXAMPLES

24698 None.

24699 APPLICATION USAGE

24700 None.

24701 RATIONALE

24702 None.

24703 FUTURE DIRECTIONS

24704 None.

24705 SEE ALSO

24706 *mbsinit()*, *mbrtowc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

24707 CHANGE HISTORY

24708 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
24709 (E).

24710 Issue 6

24711 The *mbrlen()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

24712 NAME

24713 mbrtowc — convert a character to a wide-character code (restartable)

24714 SYNOPSIS

```
24715        #include <wchar.h>
24716        size_t mbrtowc(wchar_t *restrict pwc, const char *restrict s,
24717                    size_t n, mbstate_t *restrict ps);
```

24718 DESCRIPTION

24719 CX 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.

24722 If *s* is a null pointer, the *mbrtowc()* function shall be equivalent to the call:

```
24723        mbrtowc(NULL, "", 1, ps)
```

24724 In this case, the values of the arguments *pwc* and *n* are ignored.

24725 If *s* is not a null pointer, the *mbrtowc()* function shall inspect at most *n* bytes beginning at the byte pointed to by *s* to determine the number of bytes needed to complete the next character (including any shift sequences). If the function determines that the next character is completed, it shall determine the value of the corresponding wide character and then, if *pwc* is not a null pointer, shall store that value in the object pointed to by *pwc*. If the corresponding wide character is the null wide character, the resulting state described shall be the initial conversion state.

24732 If *ps* is a null pointer, the *mbrtowc()* function shall use its own internal **mbstate_t** object, which shall be initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* shall be used to completely describe the current conversion state of the associated character sequence. The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-2001 calls *mbrtowc()*.

24737 The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

24738 RETURN VALUE

24739 The *mbrtowc()* function shall return the first of the following that applies:

24740 0	If the next <i>n</i> or fewer bytes complete the character that corresponds to the null wide character (which is the value stored).
----------------	---

24742 between 1 and *n* inclusive

24743	24744	24745	If the next <i>n</i> or fewer bytes complete a valid character (which is the value stored); the value returned shall be the number of bytes that complete the character.
-------	-------	-------	--

24746 (size_t)−2	If the next <i>n</i> bytes contribute to an incomplete but potentially valid character, and all <i>n</i> bytes have been processed (no value is stored). When <i>n</i> has at least the value of the {MB_CUR_MAX} macro, this case can only occur if <i>s</i> points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).
----------------------------------	---

24751 (size_t)−1	If an encoding error occurs, in which case the next <i>n</i> or fewer bytes do not contribute to a complete and valid character (no value is stored). In this case, [EILSEQ] shall be stored in <i>errno</i> and the conversion state is undefined.
----------------------------------	---

24754 ERRORS

24755 The *mbrtowc()* function may fail if:

24756 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

24757 [EILSEQ] Invalid character sequence is detected.

24758 EXAMPLES

24759 None.

24760 APPLICATION USAGE

24761 None.

24762 RATIONALE

24763 None.

24764 FUTURE DIRECTIONS

24765 None.

24766 SEE ALSO

24767 *mbsinit()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

24768 CHANGE HISTORY

24769 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995

24770 (E).

24771 Issue 6

24772 The *mbrtowc()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

24773 The following new requirements on POSIX implementations derive from alignment with the
24774 Single UNIX Specification:

- 24775 • The [EINVAL] error condition is added.

24776 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

24777 NAME

24778 mbsinit — determine conversion object status

24779 SYNOPSIS

24780 #include <wchar.h>

24781 int mbsinit(const mbstate_t *ps);

24782 DESCRIPTION

24783 CX 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.

24786 If *ps* is not a null pointer, the *mbsinit()* function shall determine whether the object pointed to by *ps* describes an initial conversion state.

24788 RETURN VALUE

24789 The *mbsinit()* function shall return non-zero if *ps* is a null pointer, or if the pointed-to object describes an initial conversion state; otherwise, it shall return zero.

24791 If an **mbstate_t** object is altered by any of the functions described as “restartable”, and is then used with a different character sequence, or in the other conversion direction, or with a different *LC_CTYPE* category setting than on earlier function calls, the behavior is undefined.

24794 ERRORS

24795 No errors are defined.

24796 EXAMPLES

24797 None.

24798 APPLICATION USAGE

24799 The **mbstate_t** object is used to describe the current conversion state from a particular character sequence to a wide-character sequence (or *vice versa*) under the rules of a particular setting of the *LC_CTYPE* category of the current locale.

24802 The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new character sequence in the initial shift state. A zero valued **mbstate_t** object is at least one way to describe an initial conversion state. A zero valued **mbstate_t** object can be used to initiate conversion involving any character sequence, in any *LC_CTYPE* category setting.

24806 RATIONALE

24807 None.

24808 FUTURE DIRECTIONS

24809 None.

24810 SEE ALSO

24811 *mbrlen()*, *mbtowc()*, *wcrtomb()*, *mbsrtowcs()*, *wcsrtombs()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

24813 CHANGE HISTORY

24814 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

24816 NAME

24817 mbsrtowcs — convert a character string to a wide-character string (restartable)

24818 SYNOPSIS

```
24819 #include <wchar.h>
24820 size_t mbsrtowcs(wchar_t *restrict dst, const char **restrict src,
24821           size_t len, mbstate_t *restrict ps);
```

24822 DESCRIPTION

24823 CX 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.

24826 The *mbsrtowcs()* function shall convert a sequence of characters, beginning in the conversion state described by the object pointed to by *ps*, from the array indirectly pointed to by *src* into a sequence of corresponding wide characters. If *dst* is not a null pointer, the converted characters shall be stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which shall also be stored. Conversion shall stop early in either of the following cases:

- A sequence of bytes is encountered that does not form a valid character.
- *len* codes have been stored into the array pointed to by *dst* (and *dst* is not a null pointer).

24834 Each conversion shall take place as if by a call to the *mbrtowc()* function.

24835 If *dst* is not a null pointer, the pointer object pointed to by *src* shall be assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last character converted (if any). If conversion stopped due to reaching a terminating null character, and if *dst* is not a null pointer, the resulting state described shall be the initial conversion state.

24840 If *ps* is a null pointer, the *mbsrtowcs()* function shall use its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* shall be used to completely describe the current conversion state of the associated character sequence. The implementation behaves as if no function defined in this volume of IEEE Std 1003.1-2001 calls *mbsrtowcs()*.

24845 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

24846 RETURN VALUE

24847 If the input conversion encounters a sequence of bytes that do not form a valid character, an encoding error occurs. In this case, the *mbsrtowcs()* function stores the value of the macro [EILSEQ] in *errno* and shall return (*size_t*)–1; the conversion state is undefined. Otherwise, it shall return the number of characters successfully converted, not including the terminating null (if any).

24852 ERRORS

24853 The *mbsrtowcs()* function may fail if:

24854 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

24855 [EILSEQ] Invalid character sequence is detected.

24856 EXAMPLES

24857 None.

24858 APPLICATION USAGE

24859 None.

24860 RATIONALE

24861 None.

24862 FUTURE DIRECTIONS

24863 None.

24864 SEE ALSO

24865 *mbsinit()*, *mbrtowc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

24866 CHANGE HISTORY

24867 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
24868 (E).

24869 Issue 6

24870 The *mbsrtowcs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

24871 The [EINVAL] error condition is marked CX.

24872 NAME

24873 mbstowcs — convert a character string to a wide-character string

24874 SYNOPSIS

```
24875       #include <stdlib.h>
24876
24877       size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s,
24878                    size_t n);
```

24878 DESCRIPTION

24879 CX The functionality described on this reference page is aligned with the ISO C standard. Any
24880 conflict between the requirements described here and the ISO C standard is unintentional. This
24881 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

24882 The *mbstowcs()* function shall convert a sequence of characters that begins in the initial shift
24883 state from the array pointed to by *s* into a sequence of corresponding wide-character codes and
24884 shall store not more than *n* wide-character codes into the array pointed to by *pwcs*. No
24885 characters that follow a null byte (which is converted into a wide-character code with value 0)
24886 shall be examined or converted. Each character shall be converted as if by a call to *mbtowc()*,
24887 except that the shift state of *mbtowc()* is not affected.

24888 No more than *n* elements shall be modified in the array pointed to by *pwcs*. If copying takes
24889 place between objects that overlap, the behavior is undefined.

24890 XSI The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale. If
24891 *pwcs* is a null pointer, *mbstowcs()* shall return the length required to convert the entire array
24892 regardless of the value of *n*, but no values are stored.

24893 RETURN VALUE

24894 CX If an invalid character is encountered, *mbstowcs()* shall return *(size_t)*-1 and may set *errno* to
24895 indicate the error.

24896 XSI Otherwise, *mbstowcs()* shall return the number of the array elements modified (or required if
24897 *pwcs* is null), not including a terminating 0 code, if any. The array shall not be zero-terminated if
24898 the value returned is *n*.

24899 ERRORS

24900 The *mbstowcs()* function may fail if:

24901 XSI [EILSEQ] Invalid byte sequence is detected.

24902 EXAMPLES

24903 None.

24904 APPLICATION USAGE

24905 None.

24906 RATIONALE

24907 None.

24908 FUTURE DIRECTIONS

24909 None.

24910 SEE ALSO

24911 *mblen()*, *mbtowc()*, *wctomb()*, *wcstombs()*, the Base Definitions volume of IEEE Std 1003.1-2001,
24912 <stdlib.h>

24913 CHANGE HISTORY

24914 First released in Issue 4. Aligned with the ISO C standard.

24915 Issue 6

24916 The *mbstowcs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

24917 Extensions beyond the ISO C standard are marked.

24918 NAME

24919 mbtowc — convert a character to a wide-character code

24920 SYNOPSIS

```
24921       #include <stdlib.h>
24922       int mbtowc(wchar_t *restrict pwc, const char *restrict s, size_t n);
```

24923 DESCRIPTION

24924 CX 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.

24927 If *s* is not a null pointer, *mbtowc()* shall determine the number of bytes that constitute the character pointed to by *s*. It shall then determine the wide-character code for the value of type **wchar_t** that corresponds to that character. (The value of the wide-character code corresponding to the null byte is 0.) If the character is valid and *pwc* is not a null pointer, *mbtowc()* shall store the wide-character code in the object pointed to by *pwc*.

24932 The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a state-dependent encoding, this function is placed into its initial state by a call for which its character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a null pointer shall cause this function to return a non-zero value if encodings have state dependency, and 0 otherwise. If the implementation employs special bytes to change the shift state, these bytes shall not produce separate wide-character codes, but shall be grouped with an adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be unspecified. At most *n* bytes of the array pointed to by *s* shall be examined.

24941 The implementation shall behave as if no function defined in this volume of
24942 IEEE Std 1003.1-2001 calls *mbtowc()*.

24943 RETURN VALUE

24944 If *s* is a null pointer, *mbtowc()* shall return a non-zero or 0 value, if character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mbtowc()* shall either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the converted character (if the next *n* or fewer bytes form a valid character), or return -1 and may set *errno* to indicate the error (if they do not form a valid character).

24949 In no case shall the value returned be greater than *n* or the value of the {MB_CUR_MAX} macro.

24950 ERRORS

24951 The *mbtowc()* function may fail if:

24952 XSI [EILSEQ] Invalid character sequence is detected.

24953 EXAMPLES

24954 None.

24955 APPLICATION USAGE

24956 None.

24957 RATIONALE

24958 None.

24959 FUTURE DIRECTIONS

24960 None.

24961 SEE ALSO

24962 *mblen()*, *mbstowcs()*, *wctomb()*, *wcstombs()*, the Base Definitions volume of IEEE Std 1003.1-2001,
24963 <stdlib.h>

24964 CHANGE HISTORY

24965 First released in Issue 4. Aligned with the ISO C standard.

24966 Issue 6

24967 The *mbtowc()* prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.
24968 Extensions beyond the ISO C standard are marked.

24969 NAME

24970 memccpy — copy bytes in memory

24971 SYNOPSIS

24972 XSI

```
#include <string.h>
24973     void *memccpy(void *restrict s1, const void *restrict s2,
24974         int c, size_t n);
```

24976 DESCRIPTION

24977 The *memccpy()* function shall copy bytes from memory area *s2* into *s1*, stopping after the first occurrence of byte *c* (converted to an **unsigned char**) is copied, or after *n* bytes are copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined.

24981 RETURN VALUE

24982 The *memccpy()* function shall return a pointer to the byte after the copy of *c* in *s1*, or a null pointer if *c* was not found in the first *n* bytes of *s2*.

24984 ERRORS

24985 No errors are defined.

24986 EXAMPLES

24987 None.

24988 APPLICATION USAGE

24989 The *memccpy()* function does not check for the overflow of the receiving memory area.

24990 RATIONALE

24991 None.

24992 FUTURE DIRECTIONS

24993 None.

24994 SEE ALSO

24995 The Base Definitions volume of IEEE Std 1003.1-2001, **<string.h>**

24996 CHANGE HISTORY

24997 First released in Issue 1. Derived from Issue 1 of the SVID.

24998 Issue 6

24999 The **restrict** keyword is added to the *memccpy()* prototype for alignment with the ISO/IEC 9899:1999 standard.

25001 **NAME**

25002 memchr — find byte in memory

25003 **SYNOPSIS**

25004 #include <string.h>
25005 void *memchr(const void *s, int c, size_t n);

25006 **DESCRIPTION**

25007 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25008 conflict between the requirements described here and the ISO C standard is unintentional. This
25009 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

25010 The *memchr()* function shall locate the first occurrence of *c* (converted to an **unsigned char**) in
25011 the initial *n* bytes (each interpreted as **unsigned char**) of the object pointed to by *s*.

25012 **RETURN VALUE**

25013 The *memchr()* function shall return a pointer to the located byte, or a null pointer if the byte does
25014 not occur in the object.

25015 **ERRORS**

25016 No errors are defined.

25017 **EXAMPLES**

25018 None.

25019 **APPLICATION USAGE**

25020 None.

25021 **RATIONALE**

25022 None.

25023 **FUTURE DIRECTIONS**

25024 None.

25025 **SEE ALSO**

25026 The Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

25027 **CHANGE HISTORY**

25028 First released in Issue 1. Derived from Issue 1 of the SVID.

25029 NAME

25030 **memcmp** — compare bytes in memory

25031 SYNOPSIS

```
25032       #include <string.h>
25033       int memcmp(const void *s1, const void *s2, size_t n);
```

25034 DESCRIPTION

25035 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25036 conflict between the requirements described here and the ISO C standard is unintentional. This
25037 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

25038 The *memcmp()* function shall compare the first *n* bytes (each interpreted as **unsigned char**) of the
25039 object pointed to by *s1* to the first *n* bytes of the object pointed to by *s2*.

25040 The sign of a non-zero return value shall be determined by the sign of the difference between the
25041 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the objects
25042 being compared.

25043 RETURN VALUE

25044 The *memcmp()* function shall return an integer greater than, equal to, or less than 0, if the object
25045 pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*, respectively.

25046 ERRORS

25047 No errors are defined.

25048 EXAMPLES

25049 None.

25050 APPLICATION USAGE

25051 None.

25052 RATIONALE

25053 None.

25054 FUTURE DIRECTIONS

25055 None.

25056 SEE ALSO

25057 The Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>

25058 CHANGE HISTORY

25059 First released in Issue 1. Derived from Issue 1 of the SVID.

25060 NAME

25061 `memcpy` — copy bytes in memory

25062 SYNOPSIS

25063 `#include <string.h>`
25064 `void *memcpy(void *restrict s1, const void *restrict s2, size_t n);`

25065 DESCRIPTION

25066 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25067 conflict between the requirements described here and the ISO C standard is unintentional. This
25068 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

25069 The `memcpy()` function shall copy *n* bytes from the object pointed to by *s2* into the object pointed
25070 to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

25071 RETURN VALUE

25072 The `memcpy()` function shall return *s1*; no return value is reserved to indicate an error.

25073 ERRORS

25074 No errors are defined.

25075 EXAMPLES

25076 None.

25077 APPLICATION USAGE

25078 The `memcpy()` function does not check for the overflow of the receiving memory area.

25079 RATIONALE

25080 None.

25081 FUTURE DIRECTIONS

25082 None.

25083 SEE ALSO

25084 The Base Definitions volume of IEEE Std 1003.1-2001, `<string.h>`

25085 CHANGE HISTORY

25086 First released in Issue 1. Derived from Issue 1 of the SVID.

25087 Issue 6

25088 The `memcpy()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

25089 NAME

25090 memmove — copy bytes in memory with overlapping areas

25091 SYNOPSIS

```
25092       #include <string.h>
25093       void *memmove(void *s1, const void *s2, size_t n);
```

25094 DESCRIPTION

25095 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25096 conflict between the requirements described here and the ISO C standard is unintentional. This
25097 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

25098 The *memmove()* function shall copy *n* bytes from the object pointed to by *s2* into the object
25099 pointed to by *s1*. Copying takes place as if the *n* bytes from the object pointed to by *s2* are first
25100 copied into a temporary array of *n* bytes that does not overlap the objects pointed to by *s1* and
25101 *s2*, and then the *n* bytes from the temporary array are copied into the object pointed to by *s1*.

25102 RETURN VALUE

25103 The *memmove()* function shall return *s1*; no return value is reserved to indicate an error.

25104 ERRORS

25105 No errors are defined.

25106 EXAMPLES

25107 None.

25108 APPLICATION USAGE

25109 None.

25110 RATIONALE

25111 None.

25112 FUTURE DIRECTIONS

25113 None.

25114 SEE ALSO

25115 The Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>

25116 CHANGE HISTORY

25117 First released in Issue 4. Derived from the ANSI C standard.

25118 NAME

25119 **memset** — set bytes in memory

25120 SYNOPSIS

```
25121       #include <string.h>
25122       void *memset(void *s, int c, size_t n);
```

25123 DESCRIPTION

25124 CX The functionality described on this reference page is aligned with the ISO C standard. Any
25125 conflict between the requirements described here and the ISO C standard is unintentional. This
25126 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

25127 The **memset()** function shall copy *c* (converted to an **unsigned char**) into each of the first *n* bytes
25128 of the object pointed to by *s*.

25129 RETURN VALUE

25130 The **memset()** function shall return *s*; no return value is reserved to indicate an error.

25131 ERRORS

25132 No errors are defined.

25133 EXAMPLES

25134 None.

25135 APPLICATION USAGE

25136 None.

25137 RATIONALE

25138 None.

25139 FUTURE DIRECTIONS

25140 None.

25141 SEE ALSO

25142 The Base Definitions volume of IEEE Std 1003.1-2001, **<string.h>**

25143 CHANGE HISTORY

25144 First released in Issue 1. Derived from Issue 1 of the SVID.

25145 **NAME**25146 **mkdir** — make a directory25147 **SYNOPSIS**25148

```
#include <sys/stat.h>
```



```
int mkdir(const char *path, mode_t mode);
```

25150 **DESCRIPTION**

25151 The *mkdir()* function shall create a new directory with name *path*. The file permission bits of the
25152 new directory shall be initialized from *mode*. These file permission bits of the *mode* argument
25153 shall be modified by the process' file creation mask.

25154 When bits in *mode* other than the file permission bits are set, the meaning of these additional bits
25155 is implementation-defined.

25156 The directory's user ID shall be set to the process' effective user ID. The directory's group ID
25157 shall be set to the group ID of the parent directory or to the effective group ID of the process.
25158 Implementations shall provide a way to initialize the directory's group ID to the group ID of the
25159 parent directory. Implementations may, but need not, provide an implementation-defined way
25160 to initialize the directory's group ID to the effective group ID of the calling process.

25161 The newly created directory shall be an empty directory.

25162 If *path* names a symbolic link, *mkdir()* shall fail and set *errno* to [EEXIST].

25163 Upon successful completion, *mkdir()* shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
25164 fields of the directory. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the
25165 new entry shall be marked for update.

25166 **RETURN VALUE**

25167 Upon successful completion, *mkdir()* shall return 0. Otherwise, -1 shall be returned, no directory
25168 shall be created, and *errno* shall be set to indicate the error.

25169 **ERRORS**

25170 The *mkdir()* function shall fail if:

- | | |
|-------------------------------------|---|
| 25171 [EACCES] | Search permission is denied on a component of the path prefix, or write
25172 permission is denied on the parent directory of the directory to be created. |
| 25173 [EEXIST] | The named file exists. |
| 25174 [ELOOP] | A loop exists in symbolic links encountered during resolution of the <i>path</i>
25175 argument. |
| 25176 [EMLINK] | The link count of the parent directory would exceed {LINK_MAX}. |
| 25177 [ENAMETOOLONG] | The length of the <i>path</i> argument exceeds {PATH_MAX} or a pathname
25178 component is longer than {NAME_MAX}. |
| 25180 [ENOENT] | A component of the path prefix specified by <i>path</i> does not name an existing
25181 directory or <i>path</i> is an empty string. |
| 25182 [ENOSPC] | The file system does not contain enough space to hold the contents of the new
25183 directory or to extend the parent directory of the new directory. |
| 25184 [ENOTDIR] | A component of the path prefix is not a directory. |
| 25185 [EROFS] | The parent directory resides on a read-only file system. |

25186 The *mkdir()* function may fail if:

25187 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
25188 resolution of the *path* argument.

25189 [ENAMETOOLONG]
25190 As a result of encountering a symbolic link in resolution of the *path* argument,
25191 the length of the substituted pathname string exceeded {PATH_MAX}.

25192 EXAMPLES

25193 Creating a Directory

25194 The following example shows how to create a directory named **/home/cnd/mod1**, with
25195 read/write/search permissions for owner and group, and with read/search permissions for
25196 others.

```
25197 #include <sys/types.h>
25198 #include <sys/stat.h>
25199 int status;
25200 ...
25201 status = mkdir("/home/cnd/mod1", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
```

25202 APPLICATION USAGE

25203 None.

25204 RATIONALE

25205 The *mkdir()* function originated in 4.2 BSD and was added to System V in Release 3.0.

25206 4.3 BSD detects [ENAMETOOLONG].

25207 The POSIX.1-1990 standard required that the group ID of a newly created directory be set to the
25208 group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2
25209 required that implementations provide a way to have the group ID be set to the group ID of the
25210 containing directory, but did not prohibit implementations also supporting a way to set the
25211 group ID to the effective group ID of the creating process. Conforming applications should not
25212 assume which group ID will be used. If it matters, an application can use *chown()* to set the
25213 group ID after the directory is created, or determine under what conditions the implementation
25214 will set the desired group ID.

25215 FUTURE DIRECTIONS

25216 None.

25217 SEE ALSO

25218 *umask()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/stat.h>, <sys/types.h>

25219 CHANGE HISTORY

25220 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

25221 Issue 6

25222 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

25223 The following new requirements on POSIX implementations derive from alignment with the
25224 Single UNIX Specification:

- 25225 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
25226 required for conforming implementations of previous POSIX specifications, it was not
25227 required for UNIX applications.

- 25228 • The [ELOOP] mandatory error condition is added.
- 25229 • A second [ENAMETOOLONG] is added as an optional error condition.
- 25230 The following changes were made to align with the IEEE P1003.1a draft standard:
 - 25231 • The [ELOOP] optional error condition is added.

25232 **NAME**

25233 *mkfifo* — make a FIFO special file

25234 **SYNOPSIS**

```
25235        #include <sys/stat.h>
25236        int mkfifo(const char *path, mode_t mode);
```

25237 **DESCRIPTION**

25238 The *mkfifo()* function shall create a new FIFO special file named by the pathname pointed to by *path*. The file permission bits of the new FIFO shall be initialized from *mode*. The file permission bits of the *mode* argument shall be modified by the process' file creation mask.

25241 When bits in *mode* other than the file permission bits are set, the effect is implementation-defined.

25243 If *path* names a symbolic link, *mkfifo()* shall fail and set *errno* to [EEXIST].

25244 The FIFO's user ID shall be set to the process' effective user ID. The FIFO's group ID shall be set to the group ID of the parent directory or to the effective group ID of the process. Implementations shall provide a way to initialize the FIFO's group ID to the group ID of the parent directory. Implementations may, but need not, provide an implementation-defined way to initialize the FIFO's group ID to the effective group ID of the calling process.

25249 Upon successful completion, *mkfifo()* shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry shall be marked for update.

25252 **RETURN VALUE**

25253 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned, no FIFO shall be created, and *errno* shall be set to indicate the error.

25255 **ERRORS**

25256 The *mkfifo()* function shall fail if:

25257 [EACCES] A component of the path prefix denies search permission, or write permission
25258 is denied on the parent directory of the FIFO to be created.

25259 [EEXIST] The named file already exists.

25260 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
25261 argument.

25262 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
25263 component is longer than {NAME_MAX}.

25265 [ENOENT] A component of the path prefix specified by *path* does not name an existing
25266 directory or *path* is an empty string.

25267 [ENOSPC] The directory that would contain the new file cannot be extended or the file
25268 system is out of file-allocation resources.

25269 [ENOTDIR] A component of the path prefix is not a directory.

25270 [EROFS] The named file resides on a read-only file system.

25271 The *mkfifo()* function may fail if:

25272 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
25273 resolution of the *path* argument.

25274 [ENAMETOOLONG]
25275 As a result of encountering a symbolic link in resolution of the *path* argument,
25276 the length of the substituted pathname string exceeded {PATH_MAX}.

25277 EXAMPLES

25278 Creating a FIFO File

25279 The following example shows how to create a FIFO file named /home/cnd/mod_done, with
25280 read/write permissions for owner, and with read permissions for group and others.

```
25281 #include <sys/types.h>
25282 #include <sys/stat.h>
25283
25284 int status;
25285 ...
25286 status = mkfifo("/home/cnd/mod_done", S_IWUSR | S_IRUSR |
25287 S_IRGRP | S_IROTH);
```

25287 APPLICATION USAGE

25288 None.

25289 RATIONALE

25290 The syntax of this function is intended to maintain compatibility with historical
25291 implementations of *mknod()*. The latter function was included in the 1984 /usr/group standard
25292 but only for use in creating FIFO special files. The *mknod()* function was originally excluded
25293 from the POSIX.1-1988 standard as implementation-defined and replaced by *mkdir()* and
25294 *mkfifo()*. The *mknod()* function is now included for alignment with the Single UNIX
25295 Specification.

25296 The POSIX.1-1990 standard required that the group ID of a newly created FIFO be set to the
25297 group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2
25298 required that implementations provide a way to have the group ID be set to the group ID of the
25299 containing directory, but did not prohibit implementations also supporting a way to set the
25300 group ID to the effective group ID of the creating process. Conforming applications should not
25301 assume which group ID will be used. If it matters, an application can use *chown()* to set the
25302 group ID after the FIFO is created, or determine under what conditions the implementation will
25303 set the desired group ID.

25304 FUTURE DIRECTIONS

25305 None.

25306 SEE ALSO

25307 *umask()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/stat.h>, <sys/types.h>

25308 CHANGE HISTORY

25309 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

25310 Issue 6

25311 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

25312 The following new requirements on POSIX implementations derive from alignment with the
25313 Single UNIX Specification:

- 25314 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
25315 required for conforming implementations of previous POSIX specifications, it was not
25316 required for UNIX applications.

- 25317 • The [ELOOP] mandatory error condition is added.
- 25318 • A second [ENAMETOOLONG] is added as an optional error condition.
- 25319 The following changes were made to align with the IEEE P1003.1a draft standard:
- 25320 • The [ELOOP] optional error condition is added.

25321 NAME

25322 mknod — make a directory, a special file, or a regular file

25323 SYNOPSIS

25324 XSI #include <sys/stat.h>

25325 int mknod(const char *path, mode_t mode, dev_t dev);

25326

25327 DESCRIPTION

25328 The *mknod()* function shall create a new file named by the pathname to which the argument *path* points.25330 The file type for *path* is OR'ed into the *mode* argument, and the application shall select one of the
25331 following symbolic constants:

25332

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)

25339 The only portable use of *mknod()* is to create a FIFO-special file. If *mode* is not S_IFIFO or *dev* is
2540 not 0, the behavior of *mknod()* is unspecified.25341 The permissions for the new file are OR'ed into the *mode* argument, and may be selected from
25342 any combination of the following symbolic constants:

25343

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.

25360 The user ID of the file shall be initialized to the effective user ID of the process. The group ID of
25361 the file shall be initialized to either the effective group ID of the process or the group ID of the
25362 parent directory. Implementations shall provide a way to initialize the file's group ID to the
25363 group ID of the parent directory. Implementations may, but need not, provide an
25364 implementation-defined way to initialize the file's group ID to the effective group ID of the
25365 calling process. The owner, group, and other permission bits of *mode* shall be modified by the file
25366 mode creation mask of the process. The *mknod()* function shall clear each bit whose
25367 corresponding bit in the file mode creation mask of the process is set.

25368 If *path* names a symbolic link, *mknod()* shall fail and set *errno* to [EEXIST].
25369 Upon successful completion, *mknod()* shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
25370 fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new
25371 entry shall be marked for update.

25372 Only a process with appropriate privileges may invoke *mknod()* for file types other than FIFO-
25373 special.

25374 RETURN VALUE

25375 Upon successful completion, *mknod()* shall return 0. Otherwise, it shall return -1, the new file
25376 shall not be created, and *errno* shall be set to indicate the error.

25377 ERRORS

25378 The *mknod()* function shall fail if:

25379 [EACCES] A component of the path prefix denies search permission, or write permission
25380 is denied on the parent directory.

25381 [EEXIST] The named file exists.

25382 [EINVAL] An invalid argument exists.

25383 [EIO] An I/O error occurred while accessing the file system.

25384 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
25385 argument.

25386 [ENAMETOOLONG] The length of a pathname exceeds {PATH_MAX} or a pathname component is
25387 longer than {NAME_MAX}.

25389 [ENOENT] A component of the path prefix specified by *path* does not name an existing
25390 directory or *path* is an empty string.

25391 [ENOSPC] The directory that would contain the new file cannot be extended or the file
25392 system is out of file allocation resources.

25393 [ENOTDIR] A component of the path prefix is not a directory.

25394 [EPERM] The invoking process does not have appropriate privileges and the file type is
25395 not FIFO-special.

25396 [EROFS] The directory in which the file is to be created is located on a read-only file
25397 system.

25398 The *mknod()* function may fail if:

25399 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
25400 resolution of the *path* argument.

25401 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
25402 whose length exceeds {PATH_MAX}.

25404 EXAMPLES

25405 Creating a FIFO Special File

25406 The following example shows how to create a FIFO special file named `/home/cnd/mod_done`,
25407 with read/write permissions for owner, and with read permissions for group and others.

```
25408 #include <sys/types.h>
25409 #include <sys/stat.h>
25410 dev_t dev;
25411 int status;
25412 ...
25413 status = mknod( "/home/cnd/mod_done" , S_IFIFO | S_IWUSR |
25414 S_IRUSR | S_IRGRP | S_IROTH, dev);
```

25415 APPLICATION USAGE

25416 The `mkfifo()` function is preferred over this function for making FIFO special files.

25417 RATIONALE

25418 The POSIX.1-1990 standard required that the group ID of a newly created file be set to the group
25419 ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required
25420 that implementations provide a way to have the group ID be set to the group ID of the
25421 containing directory, but did not prohibit implementations also supporting a way to set the
25422 group ID to the effective group ID of the creating process. Conforming applications should not
25423 assume which group ID will be used. If it matters, an application can use `chown()` to set the
25424 group ID after the file is created, or determine under what conditions the implementation will
25425 set the desired group ID.

25426 FUTURE DIRECTIONS

25427 None.

25428 SEE ALSO

25429 `chmod()`, `creat()`, `exec`, `mkdir()`, `mkfifo()`, `open()`, `stat()`, `umask()`, the Base Definitions volume of
25430 IEEE Std 1003.1-2001, `<sys/stat.h>`

25431 CHANGE HISTORY

25432 First released in Issue 4, Version 2.

25433 Issue 5

25434 Moved from X/OPEN UNIX extension to BASE.

25435 Issue 6

25436 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

25437 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
25438 [ELOOP] error condition is added.

25439 NAME

25440 mkstemp — make a unique filename

25441 SYNOPSIS

25442 XSI #include <stdlib.h>
25443 int mkstemp(char *template);
25444

25445 DESCRIPTION

25446 The *mkstemp()* function shall replace the contents of the string pointed to by *template* by a unique
25447 filename, and return a file descriptor for the file open for reading and writing. The function thus
25448 prevents any possible race condition between testing whether the file exists and opening it for
25449 use. The string in *template* should look like a filename with six trailing 'X's; *mkstemp()* replaces
25450 each 'X' with a character from the portable filename character set. The characters are chosen
25451 such that the resulting name does not duplicate the name of an existing file at the time of a call
25452 to *mkstemp()*.

25453 RETURN VALUE

25454 Upon successful completion, *mkstemp()* shall return an open file descriptor. Otherwise, -1 shall
25455 be returned if no suitable file could be created.

25456 ERRORS

25457 No errors are defined.

25458 EXAMPLES**25459 Generating a Filename**

25460 The following example creates a file with a 10-character name beginning with the characters
25461 "file" and opens the file for reading and writing. The value returned as the value of *fd* is a file
25462 descriptor that identifies the file.

25463 #include <stdlib.h>
25464 ...
25465 char template[] = "/tmp/fileXXXXXX";
25466 int fd;
25467 fd = mkstemp(template);

25468 APPLICATION USAGE

25469 It is possible to run out of letters.

25470 The *mkstemp()* function need not check to determine whether the filename part of *template*
25471 exceeds the maximum allowable filename length.

25472 RATIONALE

25473 None.

25474 FUTURE DIRECTIONS

25475 None.

25476 SEE ALSO

25477 *getpid()*, *open()*, *tmpfile()*, *tmpnam()*, the Base Definitions volume of IEEE Std 1003.1-2001,
25478 <stdlib.h>

25479 CHANGE HISTORY

25480 First released in Issue 4, Version 2.

25481 Issue 5

25482 Moved from X/OPEN UNIX extension to BASE.

25483 NAME

25484 mktemp — make a unique filename (LEGACY)

25485 SYNOPSIS

25486 XSI #include <stdlib.h>

25487 char *mktemp(char *template);

25488

25489 DESCRIPTION

25490 The *mktemp()* function shall replace the contents of the string pointed to by *template* by a unique
25491 filename and return *template*. The application shall initialize *template* to be a filename with six
25492 trailing 'X's; *mktemp()* shall replace each 'X' with a single byte character from the portable
25493 filename character set.

25494 RETURN VALUE

25495 The *mktemp()* function shall return the pointer *template*. If a unique name cannot be created,
25496 *template* shall point to a null string.

25497 ERRORS

25498 No errors are defined.

25499 EXAMPLES**25500 Generating a Filename**

25501 The following example replaces the contents of the "template" string with a 10-character
25502 filename beginning with the characters "file" and returns a pointer to the "template" string
25503 that contains the new filename.

```
25504        #include <stdlib.h>
25505        ...
25506        char *template = "/tmp/fileXXXXXX";
25507        char *ptr;
25508        ptr = mktemp(template);
```

25509 APPLICATION USAGE

25510 Between the time a pathname is created and the file opened, it is possible for some other process
25511 to create a file with the same name. The *mkstemp()* function avoids this problem and is preferred
25512 over this function.

25513 RATIONALE

25514 None.

25515 FUTURE DIRECTIONS

25516 This function may be withdrawn in a future version.

25517 SEE ALSO

25518 *mkstemp()*, *tmpfile()*, *tmpnam()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

25519 CHANGE HISTORY

25520 First released in Issue 4, Version 2.

25521 Issue 5

25522 Moved from X/OPEN UNIX extension to BASE.

25523 **Issue 6**

25524 This function is marked LEGACY.

25525 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

25526 NAME

25527 mktime — convert broken-down time into time since the Epoch

25528 SYNOPSIS

```
25529 #include <time.h>
25530 time_t mktime(struct tm *timeptr);
```

25531 DESCRIPTION

25532 CX 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.

25535 The *mktime()* function shall convert the broken-down time, expressed as local time, in the
 25536 structure pointed to by *timeptr*, into a time since the Epoch value with the same encoding as that
 25537 of the values returned by *time()*. The original values of the *tm_wday* and *tm_yday* components of
 25538 the structure are ignored, and the original values of the other components are not restricted to
 25539 the ranges described in <time.h>.

25540 CX A positive or 0 value for *tm_isdst* shall cause *mktime()* to presume initially that Daylight Savings
 25541 Time, respectively, is or is not in effect for the specified time. A negative value for *tm_isdst* shall
 25542 cause *mktime()* to attempt to determine whether Daylight Savings Time is in effect for the
 25543 specified time.

25544 Local timezone information shall be set as though *mktime()* called *tzset()*.

25545 The relationship between the **tm** structure (defined in the <time.h> header) and the time in
 25546 seconds since the Epoch is that the result shall be as specified in the expression given in the
 25547 definition of seconds since the Epoch (see the Base Definitions volume of IEEE Std 1003.1-2001,
 25548 Section 4.14, Seconds Since the Epoch) corrected for timezone and any seasonal time
 25549 adjustments, where the names in the structure and in the expression correspond.

25550 Upon successful completion, the values of the *tm_wday* and *tm_yday* components of the structure
 25551 shall be set appropriately, and the other components are set to represent the specified time since
 25552 the Epoch, but with their values forced to the ranges indicated in the <time.h> entry; the final
 25553 value of *tm_mday* shall not be set until *tm_mon* and *tm_year* are determined.

25554 RETURN VALUE

25555 The *mktime()* function shall return the specified time since the Epoch encoded as a value of type
 25556 **time_t**. If the time since the Epoch cannot be represented, the function shall return the value
 25557 CX (time_t)-1 and may set *errno* to indicate the error.

2
2
2

25558 ERRORS

25559 The *mktime()* function may fail if:

2

25560 CX [EOVERFLOW] The result cannot be represented.

2

25561 EXAMPLES

25562 What day of the week is July 4, 2001?

```
25563 #include <stdio.h>
25564 #include <time.h>
25565 struct tm time_str;
25566 char daybuf[20];
25567 int main(void)
25568 {
25569     time_str.tm_year = 2001 - 1900;
```

```
25570     time_str.tm_mon = 7 - 1;
25571     time_str.tm_mday = 4;
25572     time_str.tm_hour = 0;
25573     time_str.tm_min = 0;
25574     time_str.tm_sec = 1;
25575     time_str.tm_isdst = -1;
25576     if (mktime(&time_str) == -1)
25577         (void)puts("-unknown-");
25578     else {
25579         (void)strftime(daybuf, sizeof(daybuf), "%A", &time_str);
25580         (void)puts(daybuf);
25581     }
25582     return 0;
25583 }
```

25584 APPLICATION USAGE

25585 None.

25586 RATIONALE

25587 None.

25588 FUTURE DIRECTIONS

25589 None.

25590 SEE ALSO

25591 *asctime()*, *clock()*, *ctime()*, *difftime()*, *gmtime()*, *localtime()*, *strftime()*, *strptime()*, *time()*, *tzset()*, 1
25592 *utime()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**time.h**>

25593 CHANGE HISTORY

25594 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard and the ANSI C
25595 standard.

25596 Issue 6

25597 Extensions beyond the ISO C standard are marked.

25598 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/58 is applied, updating the RETURN 2
25599 VALUE and ERRORS sections to add the optional [OVERFLOW] error as a CX extension. 2

25600 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/59 is applied, adding the *tzset()* function 2
25601 to the SEE ALSO section. 2

25602 NAME

25603 mlock, munlock — lock or unlock a range of process address space (**REALTIME**)

25604 SYNOPSIS

25605 MLR #include <sys/mman.h>

```
25606 int mlock(const void *addr, size_t len);  
25607 int munlock(const void *addr, size_t len);
```

25608

25609 DESCRIPTION

25610 The *mlock()* function shall cause those whole pages containing any part of the address space of
25611 the process starting at address *addr* and continuing for *len* bytes to be memory-resident until
25612 unlocked or until the process exits or execs another process image. The implementation may
25613 require that *addr* be a multiple of {PAGESIZE}.

25614 The *munlock()* function shall unlock those whole pages containing any part of the address space
25615 of the process starting at address *addr* and continuing for *len* bytes, regardless of how many
25616 times *mlock()* has been called by the process for any of the pages in the specified range. The
25617 implementation may require that *addr* be a multiple of {PAGESIZE}.

25618 If any of the pages in the range specified to a call to *munlock()* are also mapped into the address
25619 spaces of other processes, any locks established on those pages by another process are
25620 unaffected by the call of this process to *munlock()*. If any of the pages in the range specified by a
25621 call to *munlock()* are also mapped into other portions of the address space of the calling process
25622 outside the range specified, any locks established on those pages via the other mappings are also
25623 unaffected by this call.

25624 Upon successful return from *mlock()*, pages in the specified range shall be locked and memory-
25625 resident. Upon successful return from *munlock()*, pages in the specified range shall be unlocked
25626 with respect to the address space of the process. Memory residency of unlocked pages is
25627 unspecified.

25628 The appropriate privilege is required to lock process memory with *mlock()*.

25629 RETURN VALUE

25630 Upon successful completion, the *mlock()* and *munlock()* functions shall return a value of zero.
25631 Otherwise, no change is made to any locks in the address space of the process, and the function
25632 shall return a value of -1 and set *errno* to indicate the error.

25633 ERRORS

25634 The *mlock()* and *munlock()* functions shall fail if:

25635 [ENOMEM] Some or all of the address range specified by the *addr* and *len* arguments does
25636 not correspond to valid mapped pages in the address space of the process.

25637 The *mlock()* function shall fail if:

25638 [EAGAIN] Some or all of the memory identified by the operation could not be locked
25639 when the call was made.

25640 The *mlock()* and *munlock()* functions may fail if:

25641 [EINVAL] The *addr* argument is not a multiple of {PAGESIZE}.

25642 The *mlock()* function may fail if:

25643 [ENOMEM] Locking the pages mapped by the specified range would exceed an
25644 implementation-defined limit on the amount of memory that the process may
25645 lock.

25646 [EPERM] The calling process does not have the appropriate privilege to perform the
25647 requested operation.

25648 EXAMPLES

25649 None.

25650 APPLICATION USAGE

25651 None.

25652 RATIONALE

25653 None.

25654 FUTURE DIRECTIONS

25655 None.

25656 SEE ALSO

25657 *exec*, *exit()*, *fork()*, *mlockall()*, *munmap()*, the Base Definitions volume of IEEE Std 1003.1-2001,
25658 <sys/mman.h>

25659 CHANGE HISTORY

25660 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25661 Issue 6

25662 The *mlock()* and *munlock()* functions are marked as part of the Range Memory Locking option.

25663 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25664 implementation does not support the Range Memory Locking option.

25665 NAME

25666 mlockall, munlockall — lock/unlock the address space of a process (**REALTIME**)

25667 SYNOPSIS

25668 ML #include <sys/mman.h>

25669 int mlockall(int flags);

25670 int munlockall(void);

25671

25672 DESCRIPTION

25673 The *mlockall()* function shall cause all of the pages mapped by the address space of a process to
25674 be memory-resident until unlocked or until the process exits or execs another process image. The
25675 *flags* argument determines whether the pages to be locked are those currently mapped by the
25676 address space of the process, those that are mapped in the future, or both. The *flags* argument is
25677 constructed from the bitwise-inclusive OR of one or more of the following symbolic constants,
25678 defined in <sys/mman.h>:

25679 MCL_CURRENT Lock all of the pages currently mapped into the address space of the process.

25680 MCL_FUTURE Lock all of the pages that become mapped into the address space of the
25681 process in the future, when those mappings are established.

25682 If MCL_FUTURE is specified, and the automatic locking of future mappings eventually causes
25683 the amount of locked memory to exceed the amount of available physical memory or any other
25684 implementation-defined limit, the behavior is implementation-defined. The manner in which the
25685 implementation informs the application of these situations is also implementation-defined.

25686 The *munlockall()* function shall unlock all currently mapped pages of the address space of the
25687 process. Any pages that become mapped into the address space of the process after a call to
25688 *munlockall()* shall not be locked, unless there is an intervening call to *mlockall()* specifying
25689 MCL_FUTURE or a subsequent call to *mlockall()* specifying MCL_CURRENT. If pages mapped
25690 into the address space of the process are also mapped into the address spaces of other processes
25691 and are locked by those processes, the locks established by the other processes shall be
25692 unaffected by a call by this process to *munlockall()*.

25693 Upon successful return from the *mlockall()* function that specifies MCL_CURRENT, all currently
25694 mapped pages of the process' address space shall be memory-resident and locked. Upon return
25695 from the *munlockall()* function, all currently mapped pages of the process' address space shall be
25696 unlocked with respect to the process' address space. The memory residency of unlocked pages is
25697 unspecified.

25698 The appropriate privilege is required to lock process memory with *mlockall()*.

25699 RETURN VALUE

25700 Upon successful completion, the *mlockall()* function shall return a value of zero. Otherwise, no
25701 additional memory shall be locked, and the function shall return a value of -1 and set *errno* to
25702 indicate the error. The effect of failure of *mlockall()* on previously existing locks in the address
25703 space is unspecified.

25704 If it is supported by the implementation, the *munlockall()* function shall always return a value of
25705 zero. Otherwise, the function shall return a value of -1 and set *errno* to indicate the error.

25706 ERRORS

25707 The *mlockall()* function shall fail if:

25708 [EAGAIN] Some or all of the memory identified by the operation could not be locked
25709 when the call was made.

25710	[EINVAL]	The <i>flags</i> argument is zero, or includes unimplemented flags.
25711		The <i>mlockall()</i> function may fail if:
25712	[ENOMEM]	Locking all of the pages currently mapped into the address space of the process would exceed an implementation-defined limit on the amount of memory that the process may lock.
25713		
25714		
25715	[EPERM]	The calling process does not have the appropriate privilege to perform the requested operation.
25716		

25717 EXAMPLES

25718 None.

25719 APPLICATION USAGE

25720 None.

25721 RATIONALE

25722 None.

25723 FUTURE DIRECTIONS

25724 None.

25725 SEE ALSO

25726 *exec*, *exit()*, *fork()*, *mlock()*, *munmap()*, the Base Definitions volume of IEEE Std 1003.1-2001,
25727 <sys/mman.h>

25728 CHANGE HISTORY

25729 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

25730 Issue 6

25731 The *mlockall()* and *munlockall()* functions are marked as part of the Process Memory Locking
25732 option.
25733 The [ENOSYS] error condition has been removed as stubs need not be provided if an
25734 implementation does not support the Process Memory Locking option.

25735 NAME

25736 mmap — map pages of memory

25737 SYNOPSIS

25738 MC3 #include <sys/mman.h>

1

25739 void *mmap(void *addr, size_t len, int prot, int flags,
25740 int fildes, off_t off);

25741

25742 DESCRIPTION

25743 The *mmap()* function shall establish a mapping between a process' address space and a file, shared memory object, or typed memory object. The format of the call is as follows:

25745 pa=mmap(addr, len, prot, flags, fildes, off);

25746 The *mmap()* function shall establish a mapping between the address space of the process at an
25747 address *pa* for *len* bytes to the memory object represented by the file descriptor *fildes* at offset *off*
25748 for *len* bytes. The value of *pa* is an implementation-defined function of the parameter *addr* and
25749 the values of *flags*, further described below. A successful *mmap()* call shall return *pa* as its result.
25750 The address range starting at *pa* and continuing for *len* bytes shall be legitimate for the possible
25751 (not necessarily current) address space of the process. The range of bytes starting at *off* and
25752 continuing for *len* bytes shall be legitimate for the possible (not necessarily current) offsets in the
25753 file, shared memory object, or typed memory object represented by *fildes*.25754 TYM If *fildes* represents a typed memory object opened with either the
25755 POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG
25756 flag, the memory object to be mapped shall be that portion of the typed memory object allocated
25757 by the implementation as specified below. In this case, if *off* is non-zero, the behavior of *mmap()*
25758 is undefined. If *fildes* refers to a valid typed memory object that is not accessible from the calling
25759 process, *mmap()* shall fail.25760 The mapping established by *mmap()* shall replace any previous mappings for those whole pages
25761 containing any part of the address space of the process starting at *pa* and continuing for *len*
25762 bytes.25763 If the size of the mapped file changes after the call to *mmap()* as a result of some other operation
25764 on the mapped file, the effect of references to portions of the mapped region that correspond to
25765 added or removed portions of the file is unspecified.25766 TYM The *mmap()* function shall be supported for regular files, shared memory objects, and typed
25767 memory objects. Support for any other type of file is unspecified.25768 If *len* is zero, *mmap()* shall fail and no mapping shall be established.

2

25769 The parameter *prot* determines whether read, write, execute, or some combination of accesses
25770 are permitted to the data being mapped. The *prot* shall be either PROT_NONE or the bitwise-
25771 inclusive OR of one or more of the other flags in the following table, defined in the
25772 <sys/mman.h> header.

25773
25774
25775
25776
25777
25778

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.

25779 If an implementation cannot support the combination of access types specified by *prot*, the call
25780 to *mmap()* shall fail.

25781 MPR An implementation may permit accesses other than those specified by *prot*; however, if the
25782 Memory Protection option is supported, the implementation shall not permit a write to succeed
25783 where PROT_WRITE has not been set or shall not permit any access where PROT_NONE alone
25784 has been set. The implementation shall support at least the following values of *prot*:
25785 PROT_NONE, PROT_READ, PROT_WRITE, and the bitwise-inclusive OR of PROT_READ and
25786 PROT_WRITE. If the Memory Protection option is not supported, the result of any access that
25787 conflicts with the specified protection is undefined. The file descriptor *fd* shall have been
25788 opened with read permission, regardless of the protection options specified. If PROT_WRITE is
25789 specified, the application shall ensure that it has opened the file descriptor *fd* with write
25790 permission unless MAP_PRIVATE is specified in the *flags* parameter as described below.

25791 The parameter *flags* provides other information about the handling of the mapped data. The
25792 value of *flags* is the bitwise-inclusive OR of these options, defined in <sys/mman.h>:

25793
25794
25795
25796
25797

Symbolic Constant	Description
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret <i>addr</i> exactly.

25798 Implementations that do not support the Memory Mapped Files option are not required to
25799 support MAP_PRIVATE.

25800 XSI It is implementation-defined whether MAP_FIXED shall be supported. MAP_FIXED shall be
25801 supported on XSI-conformant systems.

25802 MAP_SHARED and MAP_PRIVATE describe the disposition of write references to the memory
25803 object. If MAP_SHARED is specified, write references shall change the underlying object. If
25804 MAP_PRIVATE is specified, modifications to the mapped data by the calling process shall be
25805 visible only to the calling process and shall not change the underlying object. It is unspecified
25806 whether modifications to the underlying object done after the MAP_PRIVATE mapping is
25807 established are visible through the MAP_PRIVATE mapping. Either MAP_SHARED or
25808 MAP_PRIVATE can be specified, but not both. The mapping type is retained across *fork()*.

25809 TYM When *fd* represents a typed memory object opened with either the
25810 POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG
25811 flag, *mmap()* shall, if there are enough resources available, map *len* bytes allocated from the
25812 corresponding typed memory object which were not previously allocated to any process in any
25813 processor that may access that typed memory object. If there are not enough resources available,
25814 the function shall fail. If *fd* represents a typed memory object opened with the
25815 POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, these allocated bytes shall be contiguous
25816 within the typed memory object. If *fd* represents a typed memory object opened with the
25817 POSIX_TYPED_MEM_ALLOCATE flag, these allocated bytes may be composed of non-
25818 contiguous fragments within the typed memory object. If *fd* represents a typed memory
25819 object opened with neither the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag nor the
25820 POSIX_TYPED_MEM_ALLOCATE flag, *len* bytes starting at offset *off* within the typed memory

25821 object are mapped, exactly as when mapping a file or shared memory object. In this case, if two
25822 processes map an area of typed memory using the same *off* and *len* values and using file
25823 descriptors that refer to the same memory pool (either from the same port or from a different
25824 port), both processes shall map the same region of storage.

25825 When MAP_FIXED is set in the *flags* argument, the implementation is informed that the value of
25826 *pa* shall be *addr*, exactly. If MAP_FIXED is set, *mmap()* may return MAP_FAILED and set *errno* to
25827 [EINVAL]. If a MAP_FIXED request is successful, the mapping established by *mmap()* replaces
25828 any previous mappings for the process' pages in the range [*pa*,*pa*+*len*).

25829 When MAP_FIXED is not set, the implementation uses *addr* in an implementation-defined
25830 manner to arrive at *pa*. The *pa* so chosen shall be an area of the address space that the
25831 implementation deems suitable for a mapping of *len* bytes to the file. All implementations
25832 interpret an *addr* value of 0 as granting the implementation complete freedom in selecting *pa*,
25833 subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a
25834 process address near which the mapping should be placed. When the implementation selects a
25835 value for *pa*, it never places a mapping at address 0, nor does it replace any extant mapping.

25836 The *off* argument is constrained to be aligned and sized according to the value returned by
25837 *sysconf()* when passed _SC_PAGESIZE or _SC_PAGE_SIZE. When MAP_FIXED is specified, the
25838 application shall ensure that the argument *addr* also meets these constraints. The
25839 implementation performs mapping operations over whole pages. Thus, while the argument *len*
25840 need not meet a size or alignment constraint, the implementation shall include, in any mapping
25841 operation, any partial page specified by the range [*pa*,*pa*+*len*).

25842 The system shall always zero-fill any partial page at the end of an object. Further, the system
25843 shall never write out any modified portions of the last page of an object which are beyond its
25844 MPR end. References within the address range starting at *pa* and continuing for *len* bytes to whole
25845 pages following the end of an object shall result in delivery of a SIGBUS signal.

25846 An implementation may generate SIGBUS signals when a reference would cause an error in the
25847 mapped object, such as out-of-space condition.

25848 The *mmap()* function shall add an extra reference to the file associated with the file descriptor
25849 *fildes* which is not removed by a subsequent *close()* on that file descriptor. This reference shall be
25850 removed when there are no more mappings to the file.

25851 The *st_atime* field of the mapped file may be marked for update at any time between the *mmap()*
25852 call and the corresponding *munmap()* call. The initial read or write reference to a mapped region
25853 shall cause the file's *st_atime* field to be marked for update if it has not already been marked for
25854 update.

25855 The *st_ctime* and *st_mtime* fields of a file that is mapped with MAP_SHARED and PROT_WRITE
25856 shall be marked for update at some point in the interval between a write reference to the
25857 mapped region and the next call to *msync()* with MS_ASYNC or MS_SYNC for that portion of
25858 the file by any process. If there is no such call and if the underlying file is modified as a result of
25859 a write reference, then these fields shall be marked for update at some time after the write
25860 reference.

25861 There may be implementation-defined limits on the number of memory regions that can be
25862 mapped (per process or per system).

25863 XSI If such a limit is imposed, whether the number of memory regions that can be mapped by a
25864 process is decreased by the use of *shmat()* is implementation-defined.

25865 If *mmap()* fails for reasons other than [EBADF], [EINVAL], or [ENOTSUP], some of the
25866 mappings in the address range starting at *addr* and continuing for *len* bytes may have been
25867 unmapped.

25868 RETURN VALUE

25869 Upon successful completion, the *mmap()* function shall return the address at which the mapping
 25870 was placed (*pa*); otherwise, it shall return a value of MAP_FAILED and set *errno* to indicate the
 25871 error. The symbol MAP_FAILED is defined in the <sys/mman.h> header. No successful return
 25872 from *mmap()* shall return the value MAP_FAILED.

25873 ERRORS

25874 The *mmap()* function shall fail if:

25875	[EACCES]	The <i>fildes</i> argument is not open for read, regardless of the protection specified, 25876 or <i>fildes</i> is not open for write and PROT_WRITE was specified for a 25877 MAP_SHARED type mapping.
25878 ML	[EAGAIN]	The mapping could not be locked in memory, if required by <i>mlockall()</i> , due to 25879 a lack of resources.
25880	[EBADF]	The <i>fildes</i> argument is not a valid open file descriptor.
25881	[EINVAL]	The value of <i>len</i> is zero.
25882	[EINVAL]	The <i>addr</i> argument (if MAP_FIXED was specified) or <i>off</i> is not a multiple of 25883 the page size as returned by <i>sysconf()</i> , or is considered invalid by the 25884 implementation.
25885	[EINVAL]	The value of <i>flags</i> is invalid (neither MAP_PRIVATE nor MAP_SHARED is 25886 set).
25887	[EMFILE]	The number of mapped regions would exceed an implementation-defined 25888 limit (per process or per system).
25889	[ENODEV]	The <i>fildes</i> argument refers to a file whose type is not supported by <i>mmap()</i> .
25890	[ENOMEM]	MAP_FIXED was specified, and the range [<i>addr</i> , <i>addr+len</i>) exceeds that allowed 25891 for the address space of a process; or, if MAP_FIXED was not specified and 25892 there is insufficient room in the address space to effect the mapping.
25893 ML	[ENOMEM]	The mapping could not be locked in memory, if required by <i>mlockall()</i> , 25894 because it would require more space than the system is able to supply.
25895 TYM	[ENOMEM]	Not enough unallocated memory resources remain in the typed memory 25896 object designated by <i>fildes</i> to allocate <i>len</i> bytes.
25897	[ENOTSUP]	MAP_FIXED or MAP_PRIVATE was specified in the <i>flags</i> argument and the 25898 implementation does not support this functionality.
25899		The implementation does not support the combination of accesses requested 25900 in the <i>prot</i> argument.
25901	[ENXIO]	Addresses in the range [<i>off</i> , <i>off+len</i>) are invalid for the object specified by <i>fildes</i> .
25902	[ENXIO]	MAP_FIXED was specified in <i>flags</i> and the combination of <i>addr</i> , <i>len</i> , and <i>off</i> is 25903 invalid for the object specified by <i>fildes</i> .
25904 TYM	[ENXIO]	The <i>fildes</i> argument refers to a typed memory object that is not accessible from 25905 the calling process.
25906	[EOVERFLOW]	The file is a regular file and the value of <i>off</i> plus <i>len</i> exceeds the offset 25907 maximum established in the open file description associated with <i>fildes</i> .

2

25908 EXAMPLES

25909 None.

25910 APPLICATION USAGE

25911 Use of *mmap()* may reduce the amount of memory available to other memory allocation
25912 functions.25913 Use of MAP_FIXED may result in unspecified behavior in further use of *malloc()* and *shmat()*.
25914 The use of MAP_FIXED is discouraged, as it may prevent an implementation from making the
25915 most effective use of resources.25916 The application must ensure correct synchronization when using *mmap()* in conjunction with
25917 any other file access method, such as *read()* and *write()*, standard input/output, and *shmat()*.25918 The *mmap()* function allows access to resources via address space manipulations, instead of
25919 *read()/write()*. Once a file is mapped, all a process has to do to access it is use the data at the
25920 address to which the file was mapped. So, using pseudo-code to illustrate the way in which an
25921 existing program might be changed to use *mmap()*, the following:25922

```
fildes = open(...)  
25923 lseek(fildes, some_offset)  
25924 read(fildes, buf, len)  
25925 /* Use data in buf. */
```

25926 becomes:

25927

```
fildes = open(...)  
25928 address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)  
25929 /* Use data at address. */
```

25930 RATIONALE

25931 After considering several other alternatives, it was decided to adopt the *mmap()* definition found
25932 in SVR4 for mapping memory objects into process address spaces. The SVR4 definition is
25933 minimal, in that it describes only what has been built, and what appears to be necessary for a
25934 general and portable mapping facility.25935 Note that while *mmap()* was first designed for mapping files, it is actually a general-purpose
25936 mapping facility. It can be used to map any appropriate object, such as memory, files, devices,
25937 and so on, into the address space of a process.25938 When a mapping is established, it is possible that the implementation may need to map more
25939 than is requested into the address space of the process because of hardware requirements. An
25940 application, however, cannot count on this behavior. Implementations that do not use a paged
25941 architecture may simply allocate a common memory region and return the address of it; such
25942 implementations probably do not allocate any more than is necessary. References past the end of
25943 the requested area are unspecified.25944 If an application requests a mapping that would overlay existing mappings in the process, it
25945 might be desirable that an implementation detect this and inform the application. However, the
25946 default, portable (not MAP_FIXED) operation does not overlay existing mappings. On the other
25947 hand, if the program specifies a fixed address mapping (which requires some implementation
25948 knowledge to determine a suitable address, if the function is supported at all), then the program
25949 is presumed to be successfully managing its own address space and should be trusted when it
25950 asks to map over existing data structures. Furthermore, it is also desirable to make as few system
25951 calls as possible, and it might be considered onerous to require an *munmap()* before an *mmap()*
25952 to the same address range. This volume of IEEE Std 1003.1-2001 specifies that the new mappings
25953 replace any existing mappings, following existing practice in this regard.

25954 It is not expected, when the Memory Protection option is supported, that all hardware
25955 implementations are able to support all combinations of permissions at all addresses. When this
25956 option is supported, implementations are required to disallow write access to mappings without
25957 write permission and to disallow access to mappings without any access permission. Other than
25958 these restrictions, implementations may allow access types other than those requested by the
25959 application. For example, if the application requests only PROT_WRITE, the implementation
25960 may also allow read access. A call to *mmap()* fails if the implementation cannot support allowing
25961 all the access requested by the application. For example, some implementations cannot support
25962 a request for both write access and execute access simultaneously. All implementations
25963 supporting the Memory Protection option must support requests for no access, read access,
25964 write access, and both read and write access. Strictly conforming code must only rely on the
25965 required checks. These restrictions allow for portability across a wide range of hardware.

25966 The MAP_FIXED address treatment is likely to fail for non-page-aligned values and for certain
25967 architecture-dependent address ranges. Conforming implementations cannot count on being
25968 able to choose address values for MAP_FIXED without utilizing non-portable, implementation-
25969 defined knowledge. Nonetheless, MAP_FIXED is provided as a standard interface conforming to
25970 existing practice for utilizing such knowledge when it is available.

25971 Similarly, in order to allow implementations that do not support virtual addresses, support for
25972 directly specifying any mapping addresses via MAP_FIXED is not required and thus a
25973 conforming application may not count on it.

25974 The MAP_PRIVATE function can be implemented efficiently when memory protection hardware
25975 is available. When such hardware is not available, implementations can implement such
25976 “mappings” by simply making a real copy of the relevant data into process private memory,
25977 though this tends to behave similarly to *read()*.

25978 The function has been defined to allow for many different models of using shared memory.
25979 However, all uses are not equally portable across all machine architectures. In particular, the
25980 *mmap()* function allows the system as well as the application to specify the address at which to
25981 map a specific region of a memory object. The most portable way to use the function is always to
25982 let the system choose the address, specifying NULL as the value for the argument *addr* and not
25983 to specify MAP_FIXED.

25984 If it is intended that a particular region of a memory object be mapped at the same address in a
25985 group of processes (on machines where this is even possible), then MAP_FIXED can be used to
25986 pass in the desired mapping address. The system can still be used to choose the desired address
25987 if the first such mapping is made without specifying MAP_FIXED, and then the resulting
25988 mapping address can be passed to subsequent processes for them to pass in via MAP_FIXED.
25989 The availability of a specific address range cannot be guaranteed, in general.

25990 The *mmap()* function can be used to map a region of memory that is larger than the current size
25991 of the object. Memory access within the mapping but beyond the current end of the underlying
25992 objects may result in SIGBUS signals being sent to the process. The reason for this is that the size
25993 of the object can be manipulated by other processes and can change at any moment. The
25994 implementation should tell the application that a memory reference is outside the object where
25995 this can be detected; otherwise, written data may be lost and read data may not reflect actual
25996 data in the object.

25997 Note that references beyond the end of the object do not extend the object as the new end cannot
25998 be determined precisely by most virtual memory hardware. Instead, the size can be directly
25999 manipulated by *truncate()*.

26000 Process memory locking does apply to shared memory regions, and the MEMLOCK_FUTURE
26001 argument to *mlockall()* can be relied upon to cause new shared memory regions to be

- 26002 automatically locked.
- 26003 Existing implementations of *mmap()* return the value `-1` when unsuccessful. Since the casting of
26004 this value to type `void *` cannot be guaranteed by the ISO C standard to be distinct from a
26005 successful value, this volume of IEEE Std 1003.1-2001 defines the symbol `MAP_FAILED`, which a
26006 conforming implementation does not return as the result of a successful call.
- 26007 **FUTURE DIRECTIONS**
- 26008 None.
- 26009 **SEE ALSO**
- 26010 `exec`, `fcntl()`, `fork()`, `lockf()`, `msync()`, `munmap()`, `mprotect()`, `posix_typed_mem_open()`, `shmat()`,
26011 `sysconf()`, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/mman.h>
- 26012 **CHANGE HISTORY**
- 26013 First released in Issue 4, Version 2.
- 26014 **Issue 5**
- 26015 Moved from X/OPEN UNIX extension to BASE.
- 26016 Aligned with *mmap()* in the POSIX Realtime Extension as follows:
- 26017 • The DESCRIPTION is extensively reworded.
- 26018 • The [EAGAIN] and [ENOTSUP] mandatory error conditions are added.
- 26019 • New cases of [ENOMEM] and [ENXIO] are added as mandatory error conditions.
- 26020 • The value returned on failure is the value of the constant `MAP_FAILED`; this was previously
26021 defined as `-1`.
- 26022 Large File Summit extensions are added.
- 26023 **Issue 6**
- 26024 The *mmap()* function is marked as part of the Memory Mapped Files option.
- 26025 The Open Group Corrigendum U028/6 is applied, changing `(void *)-1` to `MAP_FAILED`.
- 26026 The following new requirements on POSIX implementations derive from alignment with the
26027 Single UNIX Specification:
- 26028 • The DESCRIPTION is updated to describe the use of `MAP_FIXED`.
- 26029 • The DESCRIPTION is updated to describe the addition of an extra reference to the file
26030 associated with the file descriptor passed to *mmap()*.
- 26031 • The DESCRIPTION is updated to state that there may be implementation-defined limits on
26032 the number of memory regions that can be mapped.
- 26033 • The DESCRIPTION is updated to describe constraints on the alignment and size of the `off`
26034 argument.
- 26035 • The [EINVAL] and [EMFILE] error conditions are added.
- 26036 • The [EOVERFLOW] error condition is added. This change is to support large files.
- 26037 The following changes are made for alignment with the ISO POSIX-1:1996 standard:
- 26038 • The DESCRIPTION is updated to describe the cases when `MAP_PRIVATE` and `MAP_FIXED`
26039 need not be supported.
- 26040 The following changes are made for alignment with IEEE Std 1003.1j-2000:

26041	• Semantics for typed memory objects are added to the DESCRIPTION.	
26042	• New [ENOMEM] and [ENXIO] errors are added to the ERRORS section.	
26043	• The <i>posix_typed_mem_open()</i> function is added to the SEE ALSO section.	
26044	The DESCRIPTION is updated to avoid use of the term “must” for application requirements.	
26045	IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/34 is applied, changing the margin code	1
26046	in the SYNOPSIS from MF SHM to MC3 (notation for MF SHM TYM).	1
26047	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/60 is applied, updating the	2
26048	DESCRIPTION and ERRORS sections to add the [EINVAL] error when <i>len</i> is zero.	2

26049 NAME

26050 modf, modff, modfl — decompose a floating-point number

26051 SYNOPSIS

```
26052 #include <math.h>
26053 double modf(double x, double *iptr);
26054 float modff(float value, float *iptr);
26055 long double modfl(long double value, long double *iptr);
```

26056 DESCRIPTION

26057 CX 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.

26060 These functions shall break the argument *x* into integral and fractional parts, each of which has
26061 the same sign as the argument. It stores the integral part as a **double** (for the *modf()* function), a
26062 **float** (for the *modff()* function), or a **long double** (for the *modfl()* function), in the object pointed
26063 to by *iptr*.

26064 RETURN VALUE

26065 Upon successful completion, these functions shall return the signed fractional part of *x*.

26066 MX If *x* is NaN, a NaN shall be returned, and **iptr* shall be set to a NaN.

26067 If *x* is ±Inf, ±0 shall be returned, and **iptr* shall be set to ±Inf.

26068 ERRORS

26069 No errors are defined.

26070 EXAMPLES

26071 None.

26072 APPLICATION USAGE

26073 The *modf()* function computes the function result and **iptr* such that:

```
26074 a = modf(x, iptr) ;
26075 x == a+*iptr ;
```

1
1

26076 allowing for the usual floating-point inaccuracies.

26077 RATIONALE

26078 None.

26079 FUTURE DIRECTIONS

26080 None.

26081 SEE ALSO

26082 *frexp()*, *isnan()*, *ldexp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

26083 CHANGE HISTORY

26084 First released in Issue 1. Derived from Issue 1 of the SVID.

26085 Issue 5

26086 The DESCRIPTION is updated to indicate how an application should check for an error. This
26087 text was previously published in the APPLICATION USAGE section.

26088 Issue 6

26089 The *modff()* and *modfl()* functions are added for alignment with the ISO/IEC 9899:1999
26090 standard.

26091 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
26092 revised to align with the ISO/IEC 9899:1999 standard.

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

26095 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/35 is applied, correcting the code example 1
26096 in the APPLICATION USAGE section. 1

26097 NAME

26098 mprotect — set protection of memory mapping

26099 SYNOPSIS

26100 MPR #include <sys/mman.h>

26101 int mprotect(void *addr, size_t len, int prot);

26102

26103 DESCRIPTION

26104 The *mprotect()* function shall change the access protections to be that specified by *prot* for those
26105 whole pages containing any part of the address space of the process starting at address *addr* and
26106 continuing for *len* bytes. The parameter *prot* determines whether read, write, execute, or some
26107 combination of accesses are permitted to the data being mapped. The *prot* argument should be
26108 either PROT_NONE or the bitwise-inclusive OR of one or more of PROT_READ, PROT_WRITE,
26109 and PROT_EXEC.

26110 If an implementation cannot support the combination of access types specified by *prot*, the call
26111 to *mprotect()* shall fail.

26112 An implementation may permit accesses other than those specified by *prot*; however, no
26113 implementation shall permit a write to succeed where PROT_WRITE has not been set or shall
26114 permit any access where PROT_NONE alone has been set. Implementations shall support at
26115 least the following values of *prot*: PROT_NONE, PROT_READ, PROT_WRITE, and the bitwise-
26116 inclusive OR of PROT_READ and PROT_WRITE. If PROT_WRITE is specified, the application
26117 shall ensure that it has opened the mapped objects in the specified address range with write
26118 permission, unless MAP_PRIVATE was specified in the original mapping, regardless of whether
26119 the file descriptors used to map the objects have since been closed.

26120 The implementation shall require that *addr* be a multiple of the page size as returned by
26121 *sysconf()*.

26122 The behavior of this function is unspecified if the mapping was not established by a call to
26123 *mmap()*.

26124 When *mprotect()* fails for reasons other than [EINVAL], the protections on some of the pages in
26125 the range [*addr*,*addr*+*len*] may have been changed.

26126 RETURN VALUE

26127 Upon successful completion, *mprotect()* shall return 0; otherwise, it shall return -1 and set *errno*
26128 to indicate the error.

26129 ERRORS

26130 The *mprotect()* function shall fail if:

26131 [EACCES] The *prot* argument specifies a protection that violates the access permission
26132 the process has to the underlying memory object.

26133 [EAGAIN] The *prot* argument specifies PROT_WRITE over a MAP_PRIVATE mapping
26134 and there are insufficient memory resources to reserve for locking the private
26135 page.

26136 [EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf()*.

26137 [ENOMEM] Addresses in the range [*addr*,*addr*+*len*] are invalid for the address space of a
26138 process, or specify one or more pages which are not mapped.

26139 [ENOMEM] The *prot* argument specifies PROT_WRITE on a MAP_PRIVATE mapping, and
26140 it would require more space than the system is able to supply for locking the
26141 private pages, if required.

26142 [ENOTSUP] The implementation does not support the combination of accesses requested
26143 in the *prot* argument.

26144 EXAMPLES

26145 None.

26146 APPLICATION USAGE

26147 The [EINVAL] error above is marked EX because it is defined as an optional error in the POSIX
26148 Realtime Extension.

26149 RATIONALE

26150 None.

26151 FUTURE DIRECTIONS

26152 None.

26153 SEE ALSO

26154 *mmap()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/mman.h>

26155 CHANGE HISTORY

26156 First released in Issue 4, Version 2.

26157 Issue 5

26158 Moved from X/OPEN UNIX extension to BASE.

26159 Aligned with *mprotect()* in the POSIX Realtime Extension as follows:

- 26160 • The DESCRIPTION is largely reworded.
- 26161 • [ENOTSUP] and a second form of [ENOMEM] are added as mandatory error conditions.
- 26162 • [EAGAIN] is moved from the optional to the mandatory error conditions.

26163 Issue 6

26164 The *mprotect()* function is marked as part of the Memory Protection option.

26165 The following new requirements on POSIX implementations derive from alignment with the
26166 Single UNIX Specification:

- 26167 • The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of
26168 the page size as returned by *sysconf()*.
- 26169 • The [EINVAL] error condition is added.

26170 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

26171 NAME

26172 *mq_close* — close a message queue (**REALTIME**)

26173 SYNOPSIS

26174 MSG *#include <mqueue.h>*

26175 *int mq_close(mqd_t mqdes);*

26176

26177 DESCRIPTION

26178 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.

26182 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.

26185 RETURN VALUE

26186 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.

26188 ERRORS

26189 The *mq_close()* function shall fail if:

26190 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

26191 EXAMPLES

26192 None.

26193 APPLICATION USAGE

26194 None.

26195 RATIONALE

26196 None.

26197 FUTURE DIRECTIONS

26198 None.

26199 SEE ALSO

26200 *mq_open()*, *mq_unlink()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the Base Definitions volume of
26201 IEEE Std 1003.1-2001, *<mqueue.h>*

26202 CHANGE HISTORY

26203 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26204 Issue 6

26205 The *mq_close()* function is marked as part of the Message Passing option.

26206 The [ENOSYS] error condition has been removed as stubs need not be provided if an
26207 implementation does not support the Message Passing option.

26208 NAME

26209 *mq_getattr* — get message queue attributes (**REALTIME**)

26210 SYNOPSIS

26211 MSG #include <mqueue.h>

26212 int *mq_getattr*(mqd_t *mqdes*, struct *mq_attr* **mqstat*);

26213

26214 DESCRIPTION

26215 The *mq_getattr*() function shall obtain status information and attributes of the message queue
26216 and the open message queue description associated with the message queue descriptor.

26217 The *mqdes* argument specifies a message queue descriptor.

26218 The results shall be returned in the **mq_attr** structure referenced by the *mqstat* argument.

26219 Upon return, the following members shall have the values associated with the open message
26220 queue description as set when the message queue was opened and as modified by subsequent
26221 *mq_setattr*() calls: *mq_flags*.

26222 The following attributes of the message queue shall be returned as set at message queue
26223 creation: *mq_maxmsg*, *mq_msgsize*.

26224 Upon return, the following members within the **mq_attr** structure referenced by the *mqstat*
26225 argument shall be set to the current state of the message queue:

26226 *mq_curmsgs* The number of messages currently on the queue.

26227 RETURN VALUE

26228 Upon successful completion, the *mq_getattr*() function shall return zero. Otherwise, the function
26229 shall return -1 and set *errno* to indicate the error.

26230 ERRORS

26231 The *mq_getattr*() function may fail if:

2

26232 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

26233 EXAMPLES

26234 None.

26235 APPLICATION USAGE

26236 None.

26237 RATIONALE

26238 None.

26239 FUTURE DIRECTIONS

26240 None.

26241 SEE ALSO

26242 *mq_open*(), *mq_send*(), *mq_setattr*(), *mq_timedsend*(), *msgctl*(), *msgget*(), *msgrcv*(), *msgsnd*(), the
26243 Base Definitions volume of IEEE Std 1003.1-2001, <mqueue.h>

26244 CHANGE HISTORY

26245 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26246 Issue 6

26247 The *mq_getattr*() function is marked as part of the Message Passing option.

26248 The [ENOSYS] error condition has been removed as stubs need not be provided if an
26249 implementation does not support the Message Passing option.

26250	The <i>mq_timedsend()</i> function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.	
26252	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/61 is applied, updating the ERRORS section to change the [EBADF] error from mandatory to optional.	2

26254 NAME

26255 *mq_notify* — notify process that a message is available (**REALTIME**)

26256 SYNOPSIS

26257 MSG #include <mqueue.h>

```
26258        int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

26259

26260 DESCRIPTION

26261 If the argument *notification* is not NULL, this function shall register the calling process to be
26262 notified of message arrival at an empty message queue associated with the specified message
26263 queue descriptor, *mqdes*. The notification specified by the *notification* argument shall be sent to
26264 the process when the message queue transitions from empty to non-empty. At any time, only
26265 one process may be registered for notification by a message queue. If the calling process or any
26266 other process has already registered for notification of message arrival at the specified message
26267 queue, subsequent attempts to register for that message queue shall fail.

26268 If *notification* is NULL and the process is currently registered for notification by the specified
26269 message queue, the existing registration shall be removed.

26270 When the notification is sent to the registered process, its registration shall be removed. The
26271 message queue shall then be available for registration.

26272 If a process has registered for notification of message arrival at a message queue and some
26273 thread is blocked in *mq_receive()* waiting to receive a message when a message arrives at the
26274 queue, the arriving message shall satisfy the appropriate *mq_receive()*. The resulting behavior is
26275 as if the message queue remains empty, and no notification shall be sent.

26276 RETURN VALUE

26277 Upon successful completion, the *mq_notify()* function shall return a value of zero; otherwise, the
26278 function shall return a value of -1 and set *errno* to indicate the error.

26279 ERRORS

26280 The *mq_notify()* function shall fail if:

26281 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

26282 [EBUSY] A process is already registered for notification by the message queue.

26283 EXAMPLES

26284 None.

26285 APPLICATION USAGE

26286 None.

26287 RATIONALE

26288 None.

26289 FUTURE DIRECTIONS

26290 None.

26291 SEE ALSO

26292 *mq_open()*, *mq_send()*, *mq_timedsend()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the Base
26293 Definitions volume of IEEE Std 1003.1-2001, <**mqueue.h**>

26294 CHANGE HISTORY

26295 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26296 Issue 6

- 26297 The *mq_notify()* function is marked as part of the Message Passing option.
- 26298 The [ENOSYS] error condition has been removed as stubs need not be provided if an
26299 implementation does not support the Message Passing option.
- 26300 The *mq_timedsend()* function is added to the SEE ALSO section for alignment with
26301 IEEE Std 1003.1d-1999.

26302 NAME

26303 mq_open — open a message queue (**REALTIME**)

26304 SYNOPSIS

26305 MSG #include <mqueue.h>

26306 mqd_t mq_open(const char *name, int oflag, ...);

26307

26308 DESCRIPTION

26309 The *mq_open()* function shall establish the connection between a process and a message queue
 26310 with a message queue descriptor. It shall create an open message queue description that refers to
 26311 the message queue, and a message queue descriptor that refers to that open message queue
 26312 description. The message queue descriptor is used by other functions to refer to that message
 26313 queue. The *name* argument points to a string naming a message queue. It is unspecified whether
 26314 the name appears in the file system and is visible to other functions that take pathnames as
 26315 arguments. The *name* argument shall conform to the construction rules for a pathname. If *name*
 26316 begins with the slash character, then processes calling *mq_open()* with the same value of *name*
 26317 shall refer to the same message queue object, as long as that name has not been removed. If *name*
 26318 does not begin with the slash character, the effect is implementation-defined. The interpretation
 26319 of slash characters other than the leading slash character in *name* is implementation-defined. If
 26320 the *name* argument is not the name of an existing message queue and creation is not requested,
 26321 *mq_open()* shall fail and return an error.

26322 A message queue descriptor may be implemented using a file descriptor, in which case
 26323 applications can open up to at least {OPEN_MAX} file and message queues.

26324 The *oflag* argument requests the desired receive and/or send access to the message queue. The
 26325 requested access permission to receive messages or send messages shall be granted if the calling
 26326 process would be granted read or write access, respectively, to an equivalently protected file.

26327 The value of *oflag* is the bitwise-inclusive OR of values from the following list. Applications
 26328 shall specify exactly one of the first three values (access modes) below in the value of *oflag*:

26329 O_RDONLY Open the message queue for receiving messages. The process can use the
 26330 returned message queue descriptor with *mq_receive()*, but not *mq_send()*. A
 26331 message queue may be open multiple times in the same or different processes
 26332 for receiving messages.

26333 O_WRONLY Open the queue for sending messages. The process can use the returned
 26334 message queue descriptor with *mq_send()* but not *mq_receive()*. A message
 26335 queue may be open multiple times in the same or different processes for
 26336 sending messages.

26337 O_RDWR Open the queue for both receiving and sending messages. The process can use
 26338 any of the functions allowed for O_RDONLY and O_WRONLY. A message
 26339 queue may be open multiple times in the same or different processes for
 26340 sending messages.

26341 Any combination of the remaining flags may be specified in the value of *oflag*:

26342 O_CREAT Create a message queue. It requires two additional arguments: *mode*, which
 26343 shall be of type **mode_t**, and *attr*, which shall be a pointer to an **mq_attr**
 26344 structure. If the pathname *name* has already been used to create a message
 26345 queue that still exists, then this flag shall have no effect, except as noted under
 26346 O_EXCL. Otherwise, a message queue shall be created without any messages
 26347 in it. The user ID of the message queue shall be set to the effective user ID of
 26348 the process, and the group ID of the message queue shall be set to the effective

26349	group ID of the process. The permission bits of the message queue shall be set to the value of the <i>mode</i> argument, except those set in the file mode creation mask of the process. When bits in <i>mode</i> other than the file permission bits are specified, the effect is unspecified. 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.	2	
26350		2	
26351		2	
26352		2	
26353		2	
26354		2	
26355		2	
26356		2	
26357		2	
26358		2	
26359		2	
26360	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.	
26361			
26362			
26363			
26364			
26365			
26366	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.	
26367			
26368			
26369	The <i>mq_open()</i> function does not add or remove messages from the queue.		
26370	RETURN VALUE		
26371	Upon successful completion, the function shall return a message queue descriptor; otherwise,		
26372	the function shall return (mqd_t)–1 and set <i>errno</i> to indicate the error.		
26373	ERRORS		
26374	The <i>mq_open()</i> function shall fail if:		
26375	[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.	
26376			
26377			
26378	[EEXIST]	O_CREAT and O_EXCL are set and the named message queue already exists.	
26379	[EINTR]	The <i>mq_open()</i> function was interrupted by a signal.	
26380	[EINVAL]	The <i>mq_open()</i> function is not supported for the given name.	
26381	[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.	
26382			
26383	[EMFILE]	Too many message queue descriptors or file descriptors are currently in use by this process.	
26384			
26385	[ENAMETOOLONG]	The length of the <i>name</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.	
26386			
26387			
26388	[ENFILE]	Too many message queues are currently open in the system.	
26389	[ENOENT]	O_CREAT is not set and the named message queue does not exist.	
26390	[ENOSPC]	There is insufficient space for the creation of the new message queue.	

26391 EXAMPLES

26392 None.

26393 APPLICATION USAGE

26394 None.

26395 RATIONALE

26396 None.

26397 FUTURE DIRECTIONS

26398 None.

26399 SEE ALSO

26400 *mq_close()*, *mq_getattr()*, *mq_receive()*, *mq_send()*, *mq_setattr()*, *mq_timedreceive()*, *mq_timedsend()*,
26401 *mq_unlink()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the Base Definitions volume of
26402 IEEE Std 1003.1-2001, <mqueue.h>

26403 CHANGE HISTORY

26404 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26405 Issue 6

26406 The *mq_open()* function is marked as part of the Message Passing option.

26407 The [ENOSYS] error condition has been removed as stubs need not be provided if an
26408 implementation does not support the Message Passing option.

26409 The *mq_timedreceive()* and *mq_timedsend()* functions are added to the SEE ALSO section for
26410 alignment with IEEE Std 1003.1d-1999.

26411 The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48.

26412 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/62 is applied, updating the description of 2
26413 the permission bits in the DESCRIPTION section. The change is made for consistency with the 2
26414 *shm_open()* and *sem_open()* functions. 2

26415 NAME

26416 mq_receive, mq_timedreceive — receive a message from a message queue (**REALTIME**)

26417 SYNOPSIS

```
26418 MSG #include <mqueue.h>
26419     ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
26420                         unsigned *msg_prio);
26421
26422 MSG TMO #include <mqueue.h>
26423     #include <time.h>
26424     ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
26425                               size_t msg_len, unsigned *restrict msg_prio,
26426                               const struct timespec *restrict abs_timeout);
26427
```

26428 DESCRIPTION

26429 The *mq_receive()* function shall receive the oldest of the highest priority message(s) from the
 26430 message queue specified by *mqdes*. If the size of the buffer in bytes, specified by the *msg_len*
 26431 argument, is less than the *mq_msgsize* attribute of the message queue, the function shall fail and
 26432 return an error. Otherwise, the selected message shall be removed from the queue and copied to
 26433 the buffer pointed to by the *msg_ptr* argument.

26434 If the value of *msg_len* is greater than {SSIZE_MAX}, the result is implementation-defined.

26435 If the argument *msg_prio* is not NULL, the priority of the selected message shall be stored in the
 26436 location referenced by *msg_prio*.

26437 If the specified message queue is empty and O_NONBLOCK is not set in the message queue
 26438 description associated with *mqdes*, *mq_receive()* shall block until a message is enqueued on the
 26439 message queue or until *mq_receive()* is interrupted by a signal. If more than one thread is waiting
 26440 to receive a message when a message arrives at an empty queue and the Priority Scheduling
 26441 option is supported, then the thread of highest priority that has been waiting the longest shall be
 26442 selected to receive the message. Otherwise, it is unspecified which waiting thread receives the
 26443 message. If the specified message queue is empty and O_NONBLOCK is set in the message
 26444 queue description associated with *mqdes*, no message shall be removed from the queue, and
 26445 *mq_receive()* shall return an error.

26446 TMO The *mq_timedreceive()* function shall receive the oldest of the highest priority messages from the
 26447 message queue specified by *mqdes* as described for the *mq_receive()* function. However, if
 26448 O_NONBLOCK was not specified when the message queue was opened via the *mq_open()*
 26449 function, and no message exists on the queue to satisfy the receive, the wait for such a message
 26450 shall be terminated when the specified timeout expires. If O_NONBLOCK is set, this function is
 26451 equivalent to *mq_receive()*.

26452 The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the
 26453 clock on which timeouts are based (that is, when the value of that clock equals or exceeds
 26454 *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time
 26455 of the call.

26456 TMO TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if
 26457 the Timers option is not supported, the timeout shall be based on the system clock as returned
 26458 by the *time()* function.

26459 TMO The resolution of the timeout shall be the resolution of the clock on which it is based. The
 26460 *timespec* argument is defined in the <time.h> header.

26461 Under no circumstance shall the operation fail with a timeout if a message can be removed from
26462 the message queue immediately. The validity of the *abs_timeout* parameter need not be checked
26463 if a message can be removed from the message queue immediately.

26464 RETURN VALUE

26465 TMO Upon successful completion, the *mq_receive()* and *mq_timedreceive()* functions shall return the
26466 length of the selected message in bytes and the message shall be removed from the queue.
26467 Otherwise, no message shall be removed from the queue, the functions shall return a value of -1,
26468 and set *errno* to indicate the error.

26469 ERRORS

26470 TMO The *mq_receive()* and *mq_timedreceive()* functions shall fail if:

26471 [EAGAIN] O_NONBLOCK was set in the message description associated with *mqdes*,
26472 and the specified message queue is empty.

26473 [EBADF] The *mqdes* argument is not a valid message queue descriptor open for reading.

26474 [EMSGSIZE] The specified message buffer size, *msg_len*, is less than the message size
26475 attribute of the message queue.

26476 TMO [EINTR] The *mq_receive()* or *mq_timedreceive()* operation was interrupted by a signal.

26477 TMO [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter
26478 specified a nanoseconds field value less than zero or greater than or equal to
26479 1 000 million.

26480 TMO [ETIMEDOUT] The O_NONBLOCK flag was not set when the message queue was opened,
26481 but no message arrived on the queue before the specified timeout expired.

26482 TMO The *mq_receive()* and *mq_timedreceive()* functions may fail if:

26483 [EBADMSG] The implementation has detected a data corruption problem with the
26484 message.

26485 EXAMPLES

26486 None.

26487 APPLICATION USAGE

26488 None.

26489 RATIONALE

26490 None.

26491 FUTURE DIRECTIONS

26492 None.

26493 SEE ALSO

26494 *mq_open()*, *mq_send()*, *mq_timedsend()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, *time()*, the Base
26495 Definitions volume of IEEE Std 1003.1-2001, <mqqueue.h>, <time.h>

26496 CHANGE HISTORY

26497 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26498 Issue 6

26499 The *mq_receive()* function is marked as part of the Message Passing option.

26500 The Open Group Corrigendum U021/4 is applied. The DESCRIPTION is changed to refer to
26501 *msg_len* rather than *maxsize*.

26502 The [ENOSYS] error condition has been removed as stubs need not be provided if an
26503 implementation does not support the Message Passing option.

26504 The following new requirements on POSIX implementations derive from alignment with the
26505 Single UNIX Specification:

- 26506 • In this function it is possible for the return value to exceed the range of the type **ssize_t** (since
26507 **size_t** has a larger range of positive values than **ssize_t**). A sentence restricting the size of
26508 the **size_t** object is added to the description to resolve this conflict.

26509 The *mq_timedreceive()* function is added for alignment with IEEE Std 1003.1d-1999.

26510 The **restrict** keyword is added to the *mq_timedreceive()* prototype for alignment with the
26511 ISO/IEC 9899: 1999 standard.

26512 IEEE PASC Interpretation 1003.1 #109 is applied, correcting the return type for *mq_timedreceive()*
26513 from **int** to **ssize_t**.

26514 NAME

26515 mq_send, mq_timedsend — send a message to a message queue (**REALTIME**)

26516 SYNOPSIS

26517 MSG #include <mqueue.h>

26518 int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
26519 unsigned msg_prio);

26520

26521 MSG TMO #include <mqueue.h>

26522 #include <time.h>

26523 int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
26524 unsigned msg_prio, const struct timespec *abs_timeout);

26525

26526 DESCRIPTION

26527 The *mq_send()* function shall add the message pointed to by the argument *msg_ptr* to the
26528 message queue specified by *mqdes*. The *msg_len* argument specifies the length of the message, in
26529 bytes, pointed to by *msg_ptr*. The value of *msg_len* shall be less than or equal to the *mq_msgsize*
26530 attribute of the message queue, or *mq_send()* shall fail.

26531 If the specified message queue is not full, *mq_send()* shall behave as if the message is inserted
26532 into the message queue at the position indicated by the *msg_prio* argument. A message with a
26533 larger numeric value of *msg_prio* shall be inserted before messages with lower values of
26534 *msg_prio*. A message shall be inserted after other messages in the queue, if any, with equal
26535 *msg_prio*. The value of *msg_prio* shall be less than {MQ_PRIO_MAX}.

26536 If the specified message queue is full and O_NONBLOCK is not set in the message queue
26537 description associated with *mqdes*, *mq_send()* shall block until space becomes available to
26538 enqueue the message, or until *mq_send()* is interrupted by a signal. If more than one thread is
26539 waiting to send when space becomes available in the message queue and the Priority Scheduling
26540 option is supported, then the thread of the highest priority that has been waiting the longest
26541 shall be unblocked to send its message. Otherwise, it is unspecified which waiting thread is
26542 unblocked. If the specified message queue is full and O_NONBLOCK is set in the message
26543 queue description associated with *mqdes*, the message shall not be queued and *mq_send()* shall
26544 return an error.

26545 TMO The *mq_timedsend()* function shall add a message to the message queue specified by *mqdes* in the
26546 manner defined for the *mq_send()* function. However, if the specified message queue is full and
26547 O_NONBLOCK is not set in the message queue description associated with *mqdes*, the wait for
26548 sufficient room in the queue shall be terminated when the specified timeout expires. If
26549 O_NONBLOCK is set in the message queue description, this function shall be equivalent to
26550 *mq_send()*.

26551 The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by
26552 the clock on which timeouts are based (that is, when the value of that clock equals or exceeds
26553 *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time
26554 of the call.

26555 TMO TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if
26556 the Timers option is not supported, the timeout shall be based on the system clock as returned
26557 by the *time()* function.

26558 TMO The resolution of the timeout shall be the resolution of the clock on which it is based. The
26559 *timespec* argument is defined in the <time.h> header.

26560 Under no circumstance shall the operation fail with a timeout if there is sufficient room in the
26561 queue to add the message immediately. The validity of the *abs_timeout* parameter need not be
26562 checked when there is sufficient room in the queue.

26563 RETURN VALUE

26564 TMO Upon successful completion, the *mq_send()* and *mq_timedsend()* functions shall return a value of
26565 zero. Otherwise, no message shall be enqueued, the functions shall return -1, and *errno* shall be
26566 set to indicate the error.

26567 ERRORS

26568 TMO The *mq_send()* and *mq_timedsend()* functions shall fail if:

26569 [EAGAIN] The O_NONBLOCK flag is set in the message queue description associated
26570 with *mqdes*, and the specified message queue is full.

26571 [EBADF] The *mqdes* argument is not a valid message queue descriptor open for writing.

26572 TMO [EINTR] A signal interrupted the call to *mq_send()* or *mq_timedsend()*.

26573 [EINVAL] The value of *msg_prio* was outside the valid range.

26574 TMO [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter
26575 specified a nanoseconds field value less than zero or greater than or equal to
26576 1 000 million.

26577 [EMSGSIZE] The specified message length, *msg_len*, exceeds the message size attribute of
26578 the message queue.

26579 TMO [ETIMEDOUT] The O_NONBLOCK flag was not set when the message queue was opened,
26580 but the timeout expired before the message could be added to the queue.

26581 EXAMPLES

26582 None.

26583 APPLICATION USAGE

26584 The value of the symbol {MQ_PRIO_MAX} limits the number of priority levels supported by the
26585 application. Message priorities range from 0 to {MQ_PRIO_MAX}-1.

26586 RATIONALE

26587 None.

26588 FUTURE DIRECTIONS

26589 None.

26590 SEE ALSO

26591 *mq_open()*, *mq_receive()*, *mq_setattr()*, *mq_timedreceive()*, *time()*, the Base Definitions volume of
26592 IEEE Std 1003.1-2001, <mqueue.h>, <time.h>

26593 CHANGE HISTORY

26594 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26595 Issue 6

26596 The *mq_send()* function is marked as part of the Message Passing option.

26597 The [ENOSYS] error condition has been removed as stubs need not be provided if an
26598 implementation does not support the Message Passing option.

26599 The *mq_timedsend()* function is added for alignment with IEEE Std 1003.1d-1999.

26600 NAME

26601 *mq_setattr* — set message queue attributes (**REALTIME**)

26602 SYNOPSIS

26603 MSG *#include <mqueue.h>*

```
26604        int mq_setattr(mqd_t mqdes, const struct mq_attr *restrict mqstat,
26605                       struct mq_attr *restrict omqstat);
```

26606

26607 DESCRIPTION

26608 The *mq_setattr()* function shall set attributes associated with the open message queue
26609 description referenced by the message queue descriptor specified by *mqdes*.

26610 The message queue attributes corresponding to the following members defined in the **mq_attr**
26611 structure shall be set to the specified values upon successful completion of *mq_setattr()*:

26612 *mq_flags* The value of this member is the bitwise-logical OR of zero or more of
26613 O_NONBLOCK and any implementation-defined flags.

26614 The values of the *mq_maxmsg*, *mq_msgsize*, and *mq_curmsgs* members of the **mq_attr** structure
26615 shall be ignored by *mq_setattr()*.

26616 If *omqstat* is non-NULL, the *mq_setattr()* function shall store, in the location referenced by
26617 *omqstat*, the previous message queue attributes and the current queue status. These values shall
26618 be the same as would be returned by a call to *mq_getattr()* at that point.

26619 RETURN VALUE

26620 Upon successful completion, the function shall return a value of zero and the attributes of the
26621 message queue shall have been changed as specified.

26622 Otherwise, the message queue attributes shall be unchanged, and the function shall return a
26623 value of -1 and set *errno* to indicate the error.

26624 ERRORS

26625 The *mq_setattr()* function shall fail if:

26626 [EBADF] The *mqdes* argument is not a valid message queue descriptor.

26627 EXAMPLES

26628 None.

26629 APPLICATION USAGE

26630 None.

26631 RATIONALE

26632 None.

26633 FUTURE DIRECTIONS

26634 None.

26635 SEE ALSO

26636 *mq_open()*, *mq_send()*, *mq_timedsend()*, *msgctl()*, *msgget()*, *msgrcv()*, *msgsnd()*, the Base
26637 Definitions volume of IEEE Std 1003.1-2001, *<mqueue.h>*

26638 CHANGE HISTORY

26639 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26640 Issue 6

- 26641 The *mq_setattr()* function is marked as part of the Message Passing option.
- 26642 The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.
- 26643
- 26644 The *mq_timedsend()* function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
- 26645
- 26646 The **restrict** keyword is added to the *mq_setattr()* prototype for alignment with the ISO/IEC 9899:1999 standard.
- 26647

26648 **NAME**26649 *mq_timedreceive* — receive a message from a message queue (**ADVANCED REALTIME**)26650 **SYNOPSIS**

```
26651 MSG TMO #include <mqqueue.h>
26652     #include <time.h>
26653     ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
26654         size_t msg_len, unsigned *restrict msg_prio,
26655         const struct timespec *restrict abs_timeout);
```

26656

26657 **DESCRIPTION**26658 Refer to *mq_receive()*.

26659 NAME

26660 **mq_timedsend** — send a message to a message queue (**ADVANCED REALTIME**)

26661 SYNOPSIS

```
26662 MSG TMO #include <mqqueue.h>
26663     #include <time.h>
26664     int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
26665             unsigned msg_prio, const struct timespec *abs_timeout);
```

26666 DESCRIPTION

26668 Refer to *mq_send()*.

26669 NAME

26670 mq_unlink — remove a message queue (**REALTIME**)

26671 SYNOPSIS

26672 MSG #include <mqueue.h>

26673 int mq_unlink(const char *name);

26674

26675 DESCRIPTION

26676 The *mq_unlink()* function shall remove the message queue named by the pathname *name*. After
26677 a successful call to *mq_unlink()* with *name*, a call to *mq_open()* with *name* shall fail if the flag
26678 O_CREAT is not set in *flags*. If one or more processes have the message queue open when
26679 *mq_unlink()* is called, destruction of the message queue shall be postponed until all references to
26680 the message queue have been closed.

26681 Calls to *mq_open()* to recreate the message queue may fail until the message queue is actually
26682 removed. However, the *mq_unlink()* call need not block until all references have been closed; it
26683 may return immediately.

26684 RETURN VALUE

26685 Upon successful completion, the function shall return a value of zero. Otherwise, the named
26686 message queue shall be unchanged by this function call, and the function shall return a value of
26687 -1 and set *errno* to indicate the error.

26688 ERRORS

26689 The *mq_unlink()* function shall fail if:

26690 [EACCES] Permission is denied to unlink the named message queue.

26691 [ENAMETOOLONG] The length of the *name* argument exceeds {PATH_MAX} or a pathname
26692 component is longer than {NAME_MAX}.

26694 [ENOENT] The named message queue does not exist.

26695 EXAMPLES

26696 None.

26697 APPLICATION USAGE

26698 None.

26699 RATIONALE

26700 None.

26701 FUTURE DIRECTIONS

26702 None.

26703 SEE ALSO

26704 *mq_close()*, *mq_open()*, *msgctl()*, *msgget()*, *msgrecv()*, *msgsnd()*, the Base Definitions volume of
26705 IEEE Std 1003.1-2001, <mqueue.h>

26706 CHANGE HISTORY

26707 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

26708 Issue 6

26709 The *mq_unlink()* function is marked as part of the Message Passing option.

26710 The Open Group Corrigendum U021/5 is applied, clarifying that upon unsuccessful completion,
26711 the named message queue is unchanged by this function.

26712
26713

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

26714 **NAME**

26715 mrand48 — generate uniformly distributed pseudo-random signed long integers

26716 **SYNOPSIS**

26717 XSI #include <stdlib.h>

26718 long mrand48(void);

26719

26720 **DESCRIPTION**26721 Refer to *drand48()*.

26722 NAME

26723 msgctl — XSI message control operations

26724 SYNOPSIS

26725 XSI #include <sys/msg.h>

```
26726 int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

26727

26728 DESCRIPTION

26729 The *msgctl()* function operates on XSI message queues (see the Base Definitions volume of
26730 IEEE Std 1003.1-2001, Section 3.224, Message Queue). It is unspecified whether this function
26731 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
26732 page 41).

26733 The *msgctl()* function shall provide message control operations as specified by *cmd*. The
26734 following values for *cmd*, and the message control operations they specify, are:

26735 IPC_STAT Place the current value of each member of the **msqid_ds** data structure
26736 associated with *msqid* into the structure pointed to by *buf*. The contents of this
26737 structure are defined in <sys/msg.h>.

26738 IPC_SET Set the value of the following members of the **msqid_ds** data structure
26739 associated with *msqid* to the corresponding value found in the structure
26740 pointed to by *buf*:

26741 msg_perm.uid
26742 msg_perm.gid
26743 msg_perm.mode
26744 msg_qbytes

26745 IPC_SET can only be executed by a process with appropriate privileges or that
26746 has an effective user ID equal to the value of **msg_perm.cuid** or
26747 **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*. Only a
26748 process with appropriate privileges can raise the value of **msg_qbytes**.

26749 IPC_RMID Remove the message queue identifier specified by *msqid* from the system and
26750 destroy the message queue and **msqid_ds** data structure associated with it.
26751 IPC_RMD can only be executed by a process with appropriate privileges or
26752 one that has an effective user ID equal to the value of **msg_perm.cuid** or
26753 **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*.

26754 RETURN VALUE

26755 Upon successful completion, *msgctl()* shall return 0; otherwise, it shall return -1 and set *errno* to
26756 indicate the error.

26757 ERRORS

26758 The *msgctl()* function shall fail if:

26759 [EACCES] The argument *cmd* is IPC_STAT and the calling process does not have read
26760 permission; see Section 2.7 (on page 39).

26761 [EINVAL] The value of *msqid* is not a valid message queue identifier; or the value of *cmd*
26762 is not a valid command.

26763 [EPERM] The argument *cmd* is IPC_RMID or IPC_SET and the effective user ID of the
26764 calling process is not equal to that of a process with appropriate privileges
26765 and it is not equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the data
26766 structure associated with *msqid*.

26767 [EPERM] The argument *cmd* is IPC_SET, an attempt is being made to increase to the
26768 value of **msg_qbytes**, and the effective user ID of the calling process does not
26769 have appropriate privileges.

26770 EXAMPLES

26771 None.

26772 APPLICATION USAGE

26773 The POSIX Realtime Extension defines alternative interfaces for interprocess communication
26774 (IPC). Application developers who need to use IPC should design their applications so that
26775 modules using the IPC routines described in Section 2.7 (on page 39) can be easily modified to
26776 use the alternative interfaces.

26777 RATIONALE

26778 None.

26779 FUTURE DIRECTIONS

26780 None.

26781 SEE ALSO

26782 Section 2.7 (on page 39), Section 2.8 (on page 41), *mq_close()*, *mq_getattr()*, *mq_notify()*,
26783 *mq_open()*, *mq_receive()*, *mq_send()*, *mq_setattr()*, *mq_unlink()*, *msgget()*, *msgrecv()*, *msgsnd()*, the
26784 Base Definitions volume of IEEE Std 1003.1-2001, <sys/msg.h>

26785 CHANGE HISTORY

26786 First released in Issue 2. Derived from Issue 2 of the SVID.

26787 Issue 5

26788 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
26789 DIRECTIONS to a new APPLICATION USAGE section.

26790 NAME

26791 msgget — get the XSI message queue identifier

26792 SYNOPSIS

26793 XSI #include <sys/msg.h>

26794 int msgget(key_t key, int msgflg);

26795

26796 DESCRIPTION

26797 The *msgget()* function operates on XSI message queues (see the Base Definitions volume of
26798 IEEE Std 1003.1-2001, Section 3.224, Message Queue). It is unspecified whether this function
26799 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
26800 page 41).

26801 The *msgget()* function shall return the message queue identifier associated with the argument
26802 *key*.

26803 A message queue identifier, associated message queue, and data structure (see <sys/msg.h>),
26804 shall be created for the argument *key* if one of the following is true:

- 26805 • The argument *key* is equal to IPC_PRIVATE.
- 26806 • The argument *key* does not already have a message queue identifier associated with it, and
(*msgflg* & IPC_CREAT) is non-zero.

26808 Upon creation, the data structure associated with the new message queue identifier shall be
26809 initialized as follows:

- 26810 • **msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** shall be set equal to the
26811 effective user ID and effective group ID, respectively, of the calling process.
- 26812 • The low-order 9 bits of **msg_perm.mode** shall be set equal to the low-order 9 bits of *msgflg*.
- 26813 • **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** shall be set equal to 0.
- 26814 • **msg_ctime** shall be set equal to the current time.
- 26815 • **msg_qbytes** shall be set equal to the system limit.

26816 RETURN VALUE

26817 Upon successful completion, *msgget()* shall return a non-negative integer, namely a message
26818 queue identifier. Otherwise, it shall return -1 and set *errno* to indicate the error.

26819 ERRORS

26820 The *msgget()* function shall fail if:

- | | |
|----------------|--|
| 26821 [EACCES] | A message queue identifier exists for the argument <i>key</i> , but operation
26822 permission as specified by the low-order 9 bits of <i>msgflg</i> would not be granted;
26823 see Section 2.7 (on page 39). |
| 26824 [EEXIST] | A message queue identifier exists for the argument <i>key</i> but ((<i>msgflg</i> &
26825 IPC_CREAT) && (<i>msgflg</i> & IPC_EXCL)) is non-zero. |
| 26826 [ENOENT] | A message queue identifier does not exist for the argument <i>key</i> and (<i>msgflg</i> &
26827 IPC_CREAT) is 0. |
| 26828 [ENOSPC] | A message queue identifier is to be created but the system-imposed limit on
26829 the maximum number of allowed message queue identifiers system-wide
26830 would be exceeded. |

26831 EXAMPLES

26832 None.

26833 APPLICATION USAGE

26834 The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). Application developers who need to use IPC should design their applications so that 26835 modules using the IPC routines described in Section 2.7 (on page 39) can be easily modified to 26836 use the alternative interfaces.

26838 RATIONALE

26839 None.

26840 FUTURE DIRECTIONS

26841 None.

26842 SEE ALSO

26843 Section 2.7 (on page 39), Section 2.8 (on page 41), *mq_close()*, *mq_getattr()*, *mq_notify()*,
26844 *mq_open()*, *mq_receive()*, *mq_send()*, *mq_setattr()*, *mq_unlink()*, *msgctl()*, *msgrcv()*, *msgsnd()*, the
26845 Base Definitions volume of IEEE Std 1003.1-2001, <sys/msg.h>

26846 CHANGE HISTORY

26847 First released in Issue 2. Derived from Issue 2 of the SVID.

26848 Issue 5

26849 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
26850 DIRECTIONS to a new APPLICATION USAGE section.

26851 NAME

26852 msgrecv — XSI message receive operation

26853 SYNOPSIS

```
26854 XSI #include <sys/msg.h>
26855 ssize_t msgrecv(int msqid, void *msgp, size_t msgsiz, long msgtyp,
26856         int msgflg);
```

26857

26858 DESCRIPTION

26859 The *msgrecv()* function operates on XSI message queues (see the Base Definitions volume of
 26860 IEEE Std 1003.1-2001, Section 3.224, Message Queue). It is unspecified whether this function
 26861 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 26862 page 41).

26863 The *msgrecv()* function shall read a message from the queue associated with the message queue
 26864 identifier specified by *msqid* and place it in the user-defined buffer pointed to by *msgp*.

26865 The application shall ensure that the argument *msgp* points to a user-defined buffer that contains
 26866 first a field of type **long** specifying the type of the message, and then a data portion that holds
 26867 the data bytes of the message. The structure below is an example of what this user-defined
 26868 buffer might look like:

```
26869 struct mymsg {
26870     long      mtype;      /* Message type. */
26871     char      mtext[1];   /* Message text. */
26872 }
```

26873 The structure member *mtype* is the received message's type as specified by the sending process.

26874 The structure member *mtext* is the text of the message.

26875 The argument *msgsz* specifies the size in bytes of *mtext*. The received message shall be truncated
 26876 to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & MSG_NOERROR) is non-zero. The
 26877 truncated part of the message shall be lost and no indication of the truncation shall be given to
 26878 the calling process.

26879 If the value of *msgsz* is greater than {SSIZE_MAX}, the result is implementation-defined.

26880 The argument *msgtyp* specifies the type of message requested as follows:

- 26881 • If *msgtyp* is 0, the first message on the queue shall be received.
- 26882 • If *msgtyp* is greater than 0, the first message of type *msgtyp* shall be received.
- 26883 • If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the
 26884 absolute value of *msgtyp* shall be received.

26885 The argument *msgflg* specifies the action to be taken if a message of the desired type is not on the
 26886 queue. These are as follows:

- 26887 • If (*msgflg* & IPC_NOWAIT) is non-zero, the calling thread shall return immediately with a
 26888 return value of -1 and *errno* set to [ENOMSG].
- 26889 • If (*msgflg* & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the
 2690 following occurs:
 - 2691 — A message of the desired type is placed on the queue.
 - 2692 — The message queue identifier *msqid* is removed from the system; when this occurs, *errno*
 2693 shall be set equal to [EIDRM] and -1 shall be returned.

26894 — The calling thread receives a signal that is to be caught; in this case a message is not
26895 received and the calling thread resumes execution in the manner prescribed in *sigaction()*.

26896 Upon successful completion, the following actions are taken with respect to the data structure
26897 associated with *msqid*:

- 26898 • **msg_qnum** shall be decremented by 1.
- 26899 • **msg_lpid** shall be set equal to the process ID of the calling process.
- 26900 • **msg_rtime** shall be set equal to the current time.

26901 RETURN VALUE

26902 Upon successful completion, *msgrecv()* shall return a value equal to the number of bytes actually
26903 placed into the buffer *mtext*. Otherwise, no message shall be received, *msgrecv()* shall return
26904 (*ssize_t*)–1, and *errno* shall be set to indicate the error.

26905 ERRORS

26906 The *msgrecv()* function shall fail if:

26907 [E2BIG]	The value of <i>mtext</i> is greater than <i>msgsz</i> and (<i>msgflg</i> & <i>MSG_NOERROR</i>) is 0.
26908 [EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 26909 39).
26910 [EIDRM]	The message queue identifier <i>msqid</i> is removed from the system.
26911 [EINTR]	The <i>msgrecv()</i> function was interrupted by a signal.
26912 [EINVAL]	<i>msqid</i> is not a valid message queue identifier.
26913 [ENOMSG]	The queue does not contain a message of the desired type and (<i>msgflg</i> & 26914 <i>IPC_NOWAIT</i>) is non-zero.

26915 EXAMPLES

26916 Receiving a Message

26917 The following example receives the first message on the queue (based on the value of the *msgtyp*
26918 argument, 0). The queue is identified by the *msqid* argument (assuming that the value has
26919 previously been set). This call specifies that an error should be reported if no message is
26920 available, but not if the message is too large. The message size is calculated directly using the
26921 *sizeof* operator.

```
26922 #include <sys/msg.h>
26923 ...
26924 int result;
26925 int msqid;
26926 struct message {
26927     long type;
26928     char text[20];
26929 } msg;
26930 long msgtyp = 0;
26931 ...
26932 result = msgrecv(msqid, (void *) &msg, sizeof(msg.text),
26933                 msgtyp, MSG_NOERROR | IPC_NOWAIT);
```

26934 APPLICATION USAGE

26935 The POSIX Realtime Extension defines alternative interfaces for interprocess communication
26936 (IPC). Application developers who need to use IPC should design their applications so that
26937 modules using the IPC routines described in Section 2.7 (on page 39) can be easily modified to
26938 use the alternative interfaces.

26939 RATIONALE

26940 None.

26941 FUTURE DIRECTIONS

26942 None.

26943 SEE ALSO

26944 Section 2.7 (on page 39), Section 2.8 (on page 41), *mq_close()*, *mq_getattr()*, *mq_notify()*,
26945 *mq_open()*, *mq_receive()*, *mq_send()*, *mq_setattr()*, *mq_unlink()*, *msgctl()*, *msgget()*, *msgsnd()*,
26946 *sigaction()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/msg.h>

26947 CHANGE HISTORY

26948 First released in Issue 2. Derived from Issue 2 of the SVID.

26949 Issue 5

26950 The type of the return value is changed from **int** to **ssize_t**, and a warning is added to the
26951 DESCRIPTION about values of *msgsz* larger than the {SSIZE_MAX}.

26952 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
26953 DIRECTIONS to the APPLICATION USAGE section.

26954 Issue 6

26955 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

26956 NAME

26957 msgsnd — XSI message send operation

26958 SYNOPSIS

26959 XSI #include <sys/msg.h>

26960 int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

26961

26962 DESCRIPTION

26963 The *msgsnd()* function operates on XSI message queues (see the Base Definitions volume of
26964 IEEE Std 1003.1-2001, Section 3.224, Message Queue). It is unspecified whether this function
26965 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
26966 page 41).

26967 The *msgsnd()* function shall send a message to the queue associated with the message queue
26968 identifier specified by *msqid*.

26969 The application shall ensure that the argument *msgp* points to a user-defined buffer that contains
26970 first a field of type **long** specifying the type of the message, and then a data portion that holds
26971 the data bytes of the message. The structure below is an example of what this user-defined
26972 buffer might look like:

```
26973 struct mymsg {  
26974     long    mtype;          /* Message type. */  
26975     char    mtext[1];        /* Message text. */  
26976 }
```

26977 The structure member *mtype* is a non-zero positive type **long** that can be used by the receiving
26978 process for message selection.

26979 The structure member *mtext* is any text of length *msgsz* bytes. The argument *msgsz* can range
26980 from 0 to a system-imposed maximum.

26981 The argument *msgflg* specifies the action to be taken if one or more of the following is true:

- 26982 • The number of bytes already on the queue is equal to **msg_qbytes**; see <sys/msg.h>.
- 26983 • The total number of messages on all queues system-wide is equal to the system-imposed
26984 limit.

26985 These actions are as follows:

- 26986 • If (*msgflg* & **IPC_NOWAIT**) is non-zero, the message shall not be sent and the calling thread
26987 shall return immediately.
- 26988 • If (*msgflg* & **IPC_NOWAIT**) is 0, the calling thread shall suspend execution until one of the
26989 following occurs:
 - 26990 — The condition responsible for the suspension no longer exists, in which case the message
26991 is sent.
 - 26992 — The message queue identifier *msqid* is removed from the system; when this occurs, *errno*
26993 shall be set equal to [EIDRM] and -1 shall be returned.
 - 26994 — The calling thread receives a signal that is to be caught; in this case the message is not
26995 sent and the calling thread resumes execution in the manner prescribed in *sigaction()*.

26996 Upon successful completion, the following actions are taken with respect to the data structure
26997 associated with *msqid*; see <sys/msg.h>:

- 26998 • **msg_qnum** shall be incremented by 1.
26999 • **msg_lspid** shall be set equal to the process ID of the calling process.
27000 • **msg_stime** shall be set equal to the current time.

27001 RETURN VALUE

27002 Upon successful completion, *msgsnd()* shall return 0; otherwise, no message shall be sent,
27003 *msgsnd()* shall return -1, and *errno* shall be set to indicate the error.

27004 ERRORS

27005 The *msgsnd()* function shall fail if:

- 27006 [**EACCES**] Operation permission is denied to the calling process; see Section 2.7 (on page
27007 39).
27008 [**EAGAIN**] The message cannot be sent for one of the reasons cited above and (*msgflg* &
27009 **IPC_NOWAIT**) is non-zero.
27010 [**EIDRM**] The message queue identifier *msqid* is removed from the system.
27011 [**EINTR**] The *msgsnd()* function was interrupted by a signal.
27012 [**EINVAL**] The value of *msqid* is not a valid message queue identifier, or the value of
27013 *mtype* is less than 1; or the value of *msgsz* is less than 0 or greater than the
27014 system-imposed limit.

27015 EXAMPLES

27016 Sending a Message

27017 The following example sends a message to the queue identified by the *msqid* argument
27018 (assuming that value has previously been set). This call specifies that an error should be
27019 reported if no message is available. The message size is calculated directly using the *sizeof*
27020 operator.

```
27021 #include <sys/msg.h>
27022 ...
27023 int result;
27024 int msqid;
27025 struct message {
27026     long type;
27027     char text[20];
27028 } msg;
27029 msg.type = 1;
27030 strcpy(msg.text, "This is message 1");
27031 ...
27032 result = msgsnd(msqid, (void *) &msg, sizeof(msg.text), IPC_NOWAIT);
```

27033 APPLICATION USAGE

27034 The POSIX Realtime Extension defines alternative interfaces for interprocess communication
27035 (IPC). Application developers who need to use IPC should design their applications so that
27036 modules using the IPC routines described in Section 2.7 (on page 39) can be easily modified to
27037 use the alternative interfaces.

27038 RATIONALE

27039 None.

27040 FUTURE DIRECTIONS

27041 None.

27042 SEE ALSO

27043 Section 2.7 (on page 39), Section 2.8 (on page 41), *mq_close()*, *mq_getattr()*, *mq_notify()*,
27044 *mq_open()*, *mq_receive()*, *mq_send()*, *mq_setattr()*, *mq_unlink()*, *msgctl()*, *msgget()*, *msgrcv()*,
27045 *sigaction()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/msg.h>

27046 CHANGE HISTORY

27047 First released in Issue 2. Derived from Issue 2 of the SVID.

27048 Issue 5

27049 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
27050 DIRECTIONS to a new APPLICATION USAGE section.

27051 Issue 6

27052 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

27053 NAME

27054 *msync* — synchronize memory with physical storage

27055 SYNOPSIS

27056 MF SIO #include <sys/mman.h>

```
27057     int msync(void *addr, size_t len, int flags);
```

27058

27059 DESCRIPTION

27060 The *msync()* function shall write all modified data to permanent storage locations, if any, in
 27061 those whole pages containing any part of the address space of the process starting at address
 27062 *addr* and continuing for *len* bytes. If no such storage exists, *msync()* need not have any effect. If
 27063 requested, the *msync()* function shall then invalidate cached copies of data.

27064 The implementation shall require that *addr* be a multiple of the page size as returned by
 27065 *sysconf()*.

27066 For mappings to files, the *msync()* function shall ensure that all write operations are completed
 27067 as defined for synchronized I/O data integrity completion. It is unspecified whether the
 27068 implementation also writes out other file attributes. When the *msync()* function is called on
 27069 MAP_PRIVATE mappings, any modified data shall not be written to the underlying object and
 27070 shall not cause such data to be made visible to other processes. It is unspecified whether data in
 27071 SHM|TYM MAP_PRIVATE mappings has any permanent storage locations. The effect of *msync()* on a
 27072 shared memory object or a typed memory object is unspecified. The behavior of this function is
 27073 unspecified if the mapping was not established by a call to *mmap()*.

27074 The *flags* argument is constructed from the bitwise-inclusive OR of one or more of the following
 27075 flags defined in the <sys/mman.h> header:

Symbolic Constant	Description
MS_ASYNC	Perform asynchronous writes.
MS_SYNC	Perform synchronous writes.
MS_INVALIDATE	Invalidate cached data.

27081 When MS_ASYNC is specified, *msync()* shall return immediately once all the write operations
 27082 are initiated or queued for servicing; when MS_SYNC is specified, *msync()* shall not return until
 27083 all write operations are completed as defined for synchronized I/O data integrity completion.
 27084 Either MS_ASYNC or MS_SYNC is specified, but not both.

27085 When MS_INVALIDATE is specified, *msync()* shall invalidate all cached copies of mapped data
 27086 that are inconsistent with the permanent storage locations such that subsequent references shall
 27087 obtain data that was consistent with the permanent storage locations sometime between the call
 27088 to *msync()* and the first subsequent memory reference to the data.

27089 If *msync()* causes any write to a file, the file's *st_ctime* and *st_mtime* fields shall be marked for
 27090 update.

27091 RETURN VALUE

27092 Upon successful completion, *msync()* shall return 0; otherwise, it shall return -1 and set *errno* to
 27093 indicate the error.

27094 ERRORS

27095 The *msync()* function shall fail if:

27096 [EBUSY] Some or all of the addresses in the range starting at *addr* and continuing for *len*
 27097 bytes are locked, and MS_INVALIDATE is specified.

27098	[EINVAL]	The value of <i>flags</i> is invalid.
27099	[EINVAL]	The value of <i>addr</i> is not a multiple of the page size {PAGESIZE}.
27100	[ENOMEM]	The addresses in the range starting at <i>addr</i> and continuing for <i>len</i> bytes are outside the range allowed for the address space of a process or specify one or more pages that are not mapped.
27101		
27102		

27103 EXAMPLES

27104 None.

27105 APPLICATION USAGE

27106 The *msync()* function is only supported if the Memory Mapped Files option and the Synchronized Input and Output option are supported, and thus need not be available on all implementations.

27109 The *msync()* function should be used by programs that require a memory object to be in a known state; for example, in building transaction facilities.

27111 Normal system activity can cause pages to be written to disk. Therefore, there are no guarantees that *msync()* is the only control over when pages are or are not written to disk.

27113 RATIONALE

27114 The *msync()* function writes out data in a mapped region to the permanent storage for the underlying object. The call to *msync()* ensures data integrity of the file.

27116 After the data is written out, any cached data may be invalidated if the MS_INVALIDATE flag was specified. This is useful on systems that do not support read/write consistency.

27118 FUTURE DIRECTIONS

27119 None.

27120 SEE ALSO

27121 *mmap()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/mman.h>

27122 CHANGE HISTORY

27123 First released in Issue 4, Version 2.

27124 Issue 5

27125 Moved from X/OPEN UNIX extension to BASE.

27126 Aligned with *msync()* in the POSIX Realtime Extension as follows:

- The DESCRIPTION is extensively reworded.
- [EBUSY] and a new form of [EINVAL] are added as mandatory error conditions.

27129 Issue 6

27130 The *msync()* function is marked as part of the Memory Mapped Files and Synchronized Input and Output options.

27132 The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- The [EBUSY] mandatory error condition is added.

27134 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of the page size.
- The second [EINVAL] error condition is made mandatory.

27139
27140

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding reference to typed memory objects.

27141 **NAME**27142 **munlock** — unlock a range of process address space27143 **SYNOPSIS**27144 MLR

```
#include <sys/mman.h>
```

27145

```
int munlock(const void *addr, size_t len);
```

27146

27147 **DESCRIPTION**27148 Refer to *mlock()*.

27149 **NAME**

27150 munlockall — unlock the address space of a process

27151 **SYNOPSIS**

27152 ML #include <sys/mman.h>
27153 int munlockall(void);
27154

27155 **DESCRIPTION**

27156 Refer to *mlockall()*.

27157 NAME

27158 munmap — unmap pages of memory

27159 SYNOPSIS

27160 MC3 #include <sys/mman.h>

1

27161 int munmap(void *addr, size_t len);

27162

27163 DESCRIPTION

27164 The *munmap()* function shall remove any mappings for those entire pages containing any part of
27165 the address space of the process starting at *addr* and continuing for *len* bytes. Further references
27166 to these pages shall result in the generation of a SIGSEGV signal to the process. If there are no
27167 mappings in the specified address range, then *munmap()* has no effect.

27168 The implementation shall require that *addr* be a multiple of the page size {PAGESIZE}.

27169 If a mapping to be removed was private, any modifications made in this address range shall be
27170 discarded.

27171 ML|MLR Any memory locks (see *mlock()* and *mlockall()*) associated with this address range shall be
27172 removed, as if by an appropriate call to *munlock()*.

27173 TYM If a mapping removed from a typed memory object causes the corresponding address range of
27174 the memory pool to be inaccessible by any process in the system except through allocatable
27175 mappings (that is, mappings of typed memory objects opened with the
27176 POSIX_TYPED_MEM_MAP_ALLOCATABLE flag), then that range of the memory pool shall
27177 become deallocated and may become available to satisfy future typed memory allocation
27178 requests.

27179 A mapping removed from a typed memory object opened with the
27180 POSIX_TYPED_MEM_MAP_ALLOCATABLE flag shall not affect in any way the availability of
27181 that typed memory for allocation.

27182 The behavior of this function is unspecified if the mapping was not established by a call to
27183 *mmap()*.

27184 RETURN VALUE

27185 Upon successful completion, *munmap()* shall return 0; otherwise, it shall return -1 and set *errno*
27186 to indicate the error.

27187 ERRORS

27188 The *munmap()* function shall fail if:

27189 [EINVAL] Addresses in the range [*addr*,*addr*+*len*] are outside the valid range for the
27190 address space of a process.

27191 [EINVAL] The *len* argument is 0.

27192 [EINVAL] The *addr* argument is not a multiple of the page size as returned by *sysconf()*.

27193 EXAMPLES

27194 None.

27195 APPLICATION USAGE

27196 The *munmap()* function is only supported if the Memory Mapped Files option or the Shared
27197 Memory Objects option is supported.

27198 RATIONALE

27199 The *munmap()* function corresponds to SVR4, just as the *mmap()* function does.

27200 It is possible that an application has applied process memory locking to a region that contains
27201 shared memory. If this has occurred, the *munmap()* call ignores those locks and, if necessary,
27202 causes those locks to be removed.

27203 FUTURE DIRECTIONS

27204 None.

27205 SEE ALSO

27206 *mlock()*, *mlockall()*, *mmap()*, *posix_typed_mem_open()*, *sysconf()*, the Base Definitions volume of
27207 IEEE Std 1003.1-2001, <signal.h>, <sys/mman.h>

27208 CHANGE HISTORY

27209 First released in Issue 4, Version 2.

27210 Issue 5

27211 Moved from X/OPEN UNIX extension to BASE.

27212 Aligned with *munmap()* in the POSIX Realtime Extension as follows:

- 27213 • The DESCRIPTION is extensively reworded.
- 27214 • The SIGBUS error is no longer permitted to be generated.

27215 Issue 6

27216 The *munmap()* function is marked as part of the Memory Mapped Files and Shared Memory
27217 Objects option.

27218 The following new requirements on POSIX implementations derive from alignment with the
27219 Single UNIX Specification:

- 27220 • The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of
27221 the page size.
- 27222 • The [EINVAL] error conditions are added.

27223 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 27224 • Semantics for typed memory objects are added to the DESCRIPTION.
- 27225 • The *posix_typed_mem_open()* function is added to the SEE ALSO section.

27226 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/36 is applied, changing the margin code 1
27227 in the SYNOPSIS from MF | SHM to MC3 (notation for MF | SHM | TYM). 1

27228 NAME

27229 nan, nanf, nanl — return quiet NaN

27230 SYNOPSIS

```
27231       #include <math.h>
27232
27233       double nan(const char *tagp);
27234       float nanf(const char *tagp);
27235       long double nanl(const char *tagp);
```

27235 DESCRIPTION

27236 CX 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.

27239 The function call *nan("n-char-sequence")* shall be equivalent to:

```
27240       strtod( "NAN(n-char-sequence)" , (char **) NULL );
```

27241 The function call *nan("")* shall be equivalent to:

```
27242       strtod( "NAN()" , (char **) NULL )
```

27243 If *tagp* does not point to an *n-char* sequence or an empty string, the function call shall be equivalent to:

```
27245       strtod( "NAN" , (char **) NULL )
```

27246 Function calls to *nanf()* and *nanl()* are equivalent to the corresponding function calls to *strtof()* and *strtold()*.

27248 RETURN VALUE

27249 These functions shall return a quiet NaN, if available, with content indicated through *tagp*.

27250 If the implementation does not support quiet NaNs, these functions shall return zero.

27251 ERRORS

27252 No errors are defined.

27253 EXAMPLES

27254 None.

27255 APPLICATION USAGE

27256 None.

27257 RATIONALE

27258 None.

27259 FUTURE DIRECTIONS

27260 None.

27261 SEE ALSO

27262 *strtod()*, *strtold()*, the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

27263 CHANGE HISTORY

27264 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

27265 NAME

27266 nanosleep — high resolution sleep (**REALTIME**)

27267 SYNOPSIS

27268 TMR #include <time.h>

```
27269 int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

27270

27271 DESCRIPTION

27272 The *nanosleep()* function shall cause the current thread to be suspended from execution until
27273 either the time interval specified by the *rqtp* argument has elapsed or a signal is delivered to the
27274 calling thread, and its action is to invoke a signal-catching function or to terminate the process.
27275 The suspension time may be longer than requested because the argument value is rounded up to
27276 an integer multiple of the sleep resolution or because of the scheduling of other activity by the
27277 system. But, except for the case of being interrupted by a signal, the suspension time shall not be
27278 less than the time specified by *rqtp*, as measured by the system clock *CLOCK_REALTIME*.

27279 The use of the *nanosleep()* function has no effect on the action or blockage of any signal.

27280 RETURN VALUE

27281 If the *nanosleep()* function returns because the requested time has elapsed, its return value shall
27282 be zero.

27283 If the *nanosleep()* function returns because it has been interrupted by a signal, it shall return a
27284 value of -1 and set *errno* to indicate the interruption. If the *rmtp* argument is non-NULL, the
27285 **timespec** structure referenced by it is updated to contain the amount of time remaining in the
27286 interval (the requested time minus the time actually slept). If the *rmtp* argument is NULL, the
27287 remaining time is not returned.

27288 If *nanosleep()* fails, it shall return a value of -1 and set *errno* to indicate the error.

27289 ERRORS

27290 The *nanosleep()* function shall fail if:

27291 [EINTR] The *nanosleep()* function was interrupted by a signal.

27292 [EINVAL] The *rqtp* argument specified a nanosecond value less than zero or greater than
27293 or equal to 1 000 million.

27294 EXAMPLES

27295 None.

27296 APPLICATION USAGE

27297 None.

27298 RATIONALE

27299 It is common to suspend execution of a thread for an interval in order to poll the status of a 2
27300 non-interrupting function. A large number of actual needs can be met with a simple extension to
27301 *sleep()* that provides finer resolution.

27302 In the POSIX.1-1990 standard and SVR4, it is possible to implement such a routine, but the
27303 frequency of wakeup is limited by the resolution of the *alarm()* and *sleep()* functions. In 4.3 BSD,
27304 it is possible to write such a routine using no static storage and reserving no system facilities.
27305 Although it is possible to write a function with similar functionality to *sleep()* using the
27306 remainder of the *timer_**() functions, such a function requires the use of signals and the
27307 reservation of some signal number. This volume of IEEE Std 1003.1-2001 requires that
27308 *nanosleep()* be non-intrusive of the signals function.

27309 The *nanosleep()* function shall return a value of 0 on success and -1 on failure or if interrupted.
27310 This latter case is different from *sleep()*. This was done because the remaining time is returned
27311 via an argument structure pointer, *rmtp*, instead of as the return value.

27312 FUTURE DIRECTIONS

27313 None.

27314 SEE ALSO

27315 *clock_nanosleep(), sleep()*, the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

27316 CHANGE HISTORY

27317 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

27318 Issue 6

27319 The `nanosleep()` function is marked as part of the Timers option.

27320 The [ENOSYS] error condition has been removed as stubs need not be provided if an
27321 implementation does not support the Timers option.

27322 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/37 is applied, updating the SEE ALSO 1
27323 section to include the *clock_nanosleep()* function. 1

27324 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/63 is applied, correcting text in the 2
27325 RATIONALE section. 2

27326 NAME

27327 nearbyint, nearbyintf, nearbyintl — floating-point rounding functions

27328 SYNOPSIS

```
27329     #include <math.h>
27330
27331     double nearbyint(double x);
27332     float nearbyintf(float x);
27333     long double nearbyintl(long double x);
```

27333 DESCRIPTION

27334 CX 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.

27337 These functions shall round their argument to an integer value in floating-point format, using the current rounding direction and without raising the inexact floating-point exception.

27339 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.

27343 RETURN VALUE

27344 Upon successful completion, these functions shall return the rounded integer value.

27345 MX If *x* is NaN, a NaN shall be returned.

27346 If *x* is ±0, ±0 shall be returned.

27347 If *x* is ±Inf, *x* shall be returned.

27348 XSI If the correct value would cause overflow, a range error shall occur and *nearbyint()*, *nearbyintf()*, and *nearbyintl()* shall return the value of the macro ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (with the same sign as *x*), respectively.

27351 ERRORS

27352 These functions shall fail if:

27353 XSI Range Error The result would cause an overflow.

27354 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero, 27355 then *errno* shall be set to [ERANGE]. If the integer expression 27356 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow 27357 floating-point exception shall be raised.

27358 EXAMPLES

27359 None.

27360 APPLICATION USAGE

27361 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & 27362 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

27363 RATIONALE

27364 None.

27365 FUTURE DIRECTIONS

27366 None.

27367 **SEE ALSO**

27368 *feclearexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18,
27369 Treatment of Error Conditions for Mathematical Functions, <math.h>

27370 **CHANGE HISTORY**

27371 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

27372 NAME

27373 nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, nexttowardl — next representable
 27374 floating-point number

27375 SYNOPSIS

```
27376 #include <math.h>
27377 double nextafter(double x, double y);
27378 float nextafterf(float x, float y);
27379 long double nextafterl(long double x, long double y);
27380 double nexttoward(double x, long double y);
27381 float nexttowardf(float x, long double y);
27382 long double nexttowardl(long double x, long double y);
```

27383 DESCRIPTION

27384 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 27385 conflict between the requirements described here and the ISO C standard is unintentional. This
 27386 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

27387 The *nextafter()*, *nextafterf()*, and *nextafterl()* functions shall compute the next representable
 27388 floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, *nextafter()* shall
 27389 return the largest representable floating-point number less than *x*. The *nextafter()*, *nextafterf()*,
 27390 and *nextafterl()* functions shall return *y* if *x* equals *y*.

27391 The *nexttoward()*, *nexttowardf()*, and *nexttowardl()* functions shall be equivalent to the
 27392 corresponding *nextafter()* functions, except that the second parameter shall have type **long**
 27393 **double** and the functions shall return *y* converted to the type of the function if *x* equals *y*.

27394 An application wishing to check for error situations should set *errno* to zero and call
 27395 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
 27396 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
 27397 zero, an error has occurred.

27398 RETURN VALUE

27399 Upon successful completion, these functions shall return the next representable floating-point
 27400 value following *x* in the direction of *y*.

27401 If *x==y*, *y* (of the type *x*) shall be returned.

27402 If *x* is finite and the correct function value would overflow, a range error shall occur and
 27403 $\pm\text{HUGE_VAL}$, $\pm\text{HUGE_VALF}$, and $\pm\text{HUGE_VALL}$ (with the same sign as *x*) shall be returned as
 27404 appropriate for the return type of the function.

27405 MX If *x* or *y* is NaN, a NaN shall be returned.

27406 If *x!=y* and the correct function value is subnormal, zero, or underflows, a range error shall
 27407 occur, and either the correct function value (if representable) or 0.0 shall be returned.

27408 ERRORS

27409 These functions shall fail if:

27410 Range Error The correct value overflows.

27411 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 27412 then *errno* shall be set to [ERANGE]. If the integer expression
 27413 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 27414 floating-point exception shall be raised.

27415 MX Range Error The correct value is subnormal or underflows.

27416 If the integer expression (`math_errhandling & MATH_ERRNO`) is non-zero,
27417 then `errno` shall be set to [ERANGE]. If the integer expression
27418 (`math_errhandling & MATH_ERREXCEPT`) is non-zero, then the underflow
27419 floating-point exception shall be raised.

27420 EXAMPLES

27421 None.

27422 APPLICATION USAGE

27423 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling &`
27424 `MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

27425 RATIONALE

27426 None.

27427 FUTURE DIRECTIONS

27428 None.

27429 SEE ALSO

27430 `feclearexcept()`, `fetestexcept()`, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18,
27431 Treatment of Error Conditions for Mathematical Functions, <**math.h**>

27432 CHANGE HISTORY

27433 First released in Issue 4, Version 2.

27434 Issue 5

27435 Moved from X/OPEN UNIX extension to BASE.

27436 Issue 6

27437 The `nextafter()` function is no longer marked as an extension.

27438 The `nextafterf()`, `nextafterl()`, `nexttoward()`, `nexttowardf()`, and `nexttowardl()` functions are added
27439 for alignment with the ISO/IEC 9899:1999 standard.

27440 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
27441 revised to align with the ISO/IEC 9899:1999 standard.

27442 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
27443 marked.

27444 NAME

27445 nftw — walk a file tree

27446 SYNOPSIS

27447 XSI #include <ftw.h>

```
27448 int nftw(const char *path, int (*fn)(const char *,
27449         const struct stat *, int, struct FTW *), int fd_limit, int flags); 2
27450
```

27451 DESCRIPTION

27452 The *nftw()* function shall recursively descend the directory hierarchy rooted in *path*. The *nftw()* function has a similar effect to *ftw()* except that it takes an additional argument *flags*, which is a bitwise-inclusive OR of zero or more of the following flags:

27455 FTW_CHDIR If set, *nftw()* shall change the current working directory to each directory as it
27456 reports files in that directory. If clear, *nftw()* shall not change the current
27457 working directory.

27458 FTW_DEPTH If set, *nftw()* shall report all files in a directory before reporting the directory
27459 itself. If clear, *nftw()* shall report any directory before reporting the files in that
27460 directory.

27461 FTW_MOUNT If set, *nftw()* shall only report files in the same file system as *path*. If clear,
27462 *nftw()* shall report all files encountered during the walk.

27463 FTW_PHYS If set, *nftw()* shall perform a physical walk and shall not follow symbolic links.
27464 If FTW_PHYS is clear and FTW_DEPTH is set, *nftw()* shall follow links instead of reporting
27465 them, but shall not report any directory that would be a descendant of itself. If FTW_PHYS is
27466 clear and FTW_DEPTH is clear, *nftw()* shall follow links instead of reporting them, but shall not
27467 report the contents of any directory that would be a descendant of itself.

27468 At each file it encounters, *nftw()* shall call the user-supplied function *fn* with four arguments:

- 27469 • The first argument is the pathname of the object.
- 27470 • The second argument is a pointer to the **stat** buffer containing information on the object.
- 27471 • The third argument is an integer giving additional information. Its value is one of the
27472 following:

27473 FTW_F The object is a file.

27474 FTW_D The object is a directory.

27475 FTW_DP The object is a directory and subdirectories have been visited. (This condition
27476 shall only occur if the FTW_DEPTH flag is included in *flags*.)

27477 FTW_SL The object is a symbolic link. (This condition shall only occur if the FTW_PHYS
27478 flag is included in *flags*.)

27479 FTW_SLN The object is a symbolic link that does not name an existing file. (This
27480 condition shall only occur if the FTW_PHYS flag is not included in *flags*.)

27481 FTW_DNR The object is a directory that cannot be read. The *fn* function shall not be called
27482 for any of its descendants.

27483 FTW_NS The **stat()** function failed on the object because of lack of appropriate
27484 permission. The **stat** buffer passed to *fn* is undefined. Failure of **stat()** for any
27485 other reason is considered an error and *nftw()* shall return -1.

- 27486 • The fourth argument is a pointer to an **FTW** structure. The value of **base** is the offset of the
27487 object's filename in the pathname passed as the first argument to *fn*. The value of **level**
27488 indicates depth relative to the root of the walk, where the root level is 0.

27489 The results are unspecified if the application-supplied *fn* function does not preserve the current
27490 working directory.

27491 The argument *fd_limit* sets the maximum number of file descriptors that shall be used by *nftw()* 2
27492 while traversing the file tree. At most one file descriptor shall be used for each directory level.

27493 The *nftw()* function need not be reentrant. A function that is not required to be reentrant is not
27494 required to be thread-safe.

27495 RETURN VALUE

27496 The *nftw()* function shall continue until the first of the following conditions occurs:

- 27497 • An invocation of *fn* shall return a non-zero value, in which case *nftw()* shall return that value.
- 27498 • The *nftw()* function detects an error other than [EACCES] (see FTW_DNR and FTW_NS
27499 above), in which case *nftw()* shall return -1 and set *errno* to indicate the error.
- 27500 • The tree is exhausted, in which case *nftw()* shall return 0.

27501 ERRORS

27502 The *nftw()* function shall fail if:

- 27503 [EACCES] Search permission is denied for any component of *path* or read permission is
27504 denied for *path*, or *fn* returns -1 and does not reset *errno*.
- 27505 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
27506 argument.
- 27507 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
27508 component is longer than {NAME_MAX}.
- 27509 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.
- 27510 [ENOTDIR] A component of *path* is not a directory.
- 27511 [EOVERFLOW] A field in the **stat** structure cannot be represented correctly in the current
27512 programming environment for one or more files found in the file hierarchy.

27513 The *nftw()* function may fail if:

- 27514 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
27515 resolution of the *path* argument.
- 27516 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.
- 27517 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
27518 whose length exceeds {PATH_MAX}.
- 27519 [ENFILE] Too many files are currently open in the system.

27520 In addition, *errno* may be set if the function pointed to by *fn* causes *errno* to be set.

27523 EXAMPLES

27524 The following example walks the `/tmp` directory and its subdirectories, calling the `nftw()` 2
27525 function for every directory entry, using a maximum of 5 file descriptors. 2

```
27526     #include <ftw.h>
27527     ...
27528     int nftwfunc(const char *, const struct stat *, int, struct FTW *);
27529     int nftwfunc(const char *filename, const struct stat *statptr,
27530                int fileflags, struct FTW *pfwt)
27531     {
27532         return 0;
27533     }
27534     ...
27535     char *startpath = "/tmp";
27536     int fd_limit = 5; 2
27537     int flags = FTW_CHDIR | FTW_DEPTH | FTW_MOUNT; 2
27538     int ret;
27539     ret = nftw(startpath, nftwfunc, fd_limit, flags); 2
```

27540 APPLICATION USAGE

27541 None.

27542 RATIONALE

27543 None.

27544 FUTURE DIRECTIONS

27545 None.

27546 SEE ALSO

27547 `lstat()`, `opendir()`, `readdir()`, `stat()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<ftw.h>`

27548 CHANGE HISTORY

27549 First released in Issue 4, Version 2.

27550 Issue 5

27551 Moved from X/OPEN UNIX extension to BASE.

27552 In the DESCRIPTION, the definition of the *depth* argument is clarified.

27553 Issue 6

27554 The Open Group Base Resolution bwg97-003 is applied.

27555 The ERRORS section is updated as follows:

- 27556 • The wording of the mandatory [ELOOP] error condition is updated.
- 27557 • A second optional [ELOOP] error condition is added.
- 27558 • The [EOVERFLOW] mandatory error condition is added.

27559 Text is added to the DESCRIPTION to say that the `nftw()` function need not be reentrant and
27560 that the results are unspecified if the application-supplied *fn* function does not preserve the
27561 current working directory.

27562 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/64 is applied, changing the argument 2
27563 *depth* to *fd_limit* throughout and changing “to a maximum of 5 levels deep” to “using a 2
27564 maximum of 5 file descriptors” in the EXAMPLES section. 2

27565 NAME

27566 nice — change the nice value of a process

27567 SYNOPSIS

27568 XSI

```
#include <unistd.h>
int nice(int incr);
```

27571 DESCRIPTION

27572 The *nice()* function shall add the value of *incr* to the nice value of the calling process. A process'
27573 nice value is a non-negative number for which a more positive value shall result in less favorable
27574 scheduling.

27575 A maximum nice value of $2^{\{NZERO\}} - 1$ and a minimum nice value of 0 shall be imposed by the
27576 system. Requests for values above or below these limits shall result in the nice value being set to
27577 the corresponding limit. Only a process with appropriate privileges can lower the nice value.

27578 PS|TPS Calling the *nice()* function has no effect on the priority of processes or threads with policy
27579 SCHED_FIFO or SCHED_RR. The effect on processes or threads with other scheduling policies
27580 is implementation-defined.

27581 The nice value set with *nice()* shall be applied to the process. If the process is multi-threaded, the
27582 nice value shall affect all system scope threads in the process.

27583 As -1 is a permissible return value in a successful situation, an application wishing to check for
27584 error situations should set *errno* to 0, then call *nice()*, and if it returns -1, check to see whether
27585 *errno* is non-zero.

27586 RETURN VALUE

27587 Upon successful completion, *nice()* shall return the new nice value -{NZERO}. Otherwise, -1
27588 shall be returned, the process' nice value shall not be changed, and *errno* shall be set to indicate
27589 the error.

27590 ERRORS

27591 The *nice()* function shall fail if:

27592 [EPERM] The *incr* argument is negative and the calling process does not have
27593 appropriate privileges.

27594 EXAMPLES**27595 Changing the Nice Value**

27596 The following example adds the value of the *incr* argument, -20, to the nice value of the calling
27597 process.

```
27598 #include <unistd.h>
27599 ...
27600 int incr = -20;
27601 int ret;
27602 ret = nice(incr);
```

27603 APPLICATION USAGE

27604 None.

27605 RATIONALE

27606 None.

27607 FUTURE DIRECTIONS

27608 None.

27609 SEE ALSO

27610 *getpriority()*, *setpriority()*, the Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>,
27611 <unistd.h>

27612 CHANGE HISTORY

27613 First released in Issue 1. Derived from Issue 1 of the SVID.

27614 Issue 5

27615 A statement is added to the description indicating the effects of this function on the different
27616 scheduling policies and multi-threaded processes.

27617 **NAME**

27618 nl_langinfo — language information

27619 **SYNOPSIS**

27620 XSI #include <langinfo.h>

27621 char *nl_langinfo(nl_item item);

27622

27623 **DESCRIPTION**

27624 The *nl_langinfo()* function shall return a pointer to a string containing information relevant to
27625 the particular language or cultural area defined in the program's locale (see <**langinfo.h**>). The
27626 manifest constant names and values of *item* are defined in <**langinfo.h**>. For example:

27627 nl_langinfo(ABDAY_1)

27628 would return a pointer to the string "Dom" if the identified language was Portuguese, and
27629 "Sun" if the identified language was English.

27630 Calls to *setlocale()* with a category corresponding to the category of *item* (see <**langinfo.h**>), or to
27631 the category *LC_ALL*, may overwrite the array pointed to by the return value.

27632 The *nl_langinfo()* function need not be reentrant. A function that is not required to be reentrant is
27633 not required to be thread-safe.

27634 **RETURN VALUE**

27635 In a locale where *langinfo* data is not defined, *nl_langinfo()* shall return a pointer to the
27636 corresponding string in the POSIX locale. In all locales, *nl_langinfo()* shall return a pointer to an
27637 empty string if *item* contains an invalid setting.

27638 This pointer may point to static data that may be overwritten on the next call.

27639 **ERRORS**

27640 No errors are defined.

27641 **EXAMPLES**27642 **Getting Date and Time Formatting Information**

27643 The following example returns a pointer to a string containing date and time formatting
27644 information, as defined in the *LC_TIME* category of the current locale.

```
27645 #include <time.h>
27646 #include <langinfo.h>
27647 ...
27648 strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);
27649 ...
```

27650 **APPLICATION USAGE**

27651 The array pointed to by the return value should not be modified by the program, but may be
27652 modified by further calls to *nl_langinfo()*.

27653 **RATIONALE**

27654 None.

27655 **FUTURE DIRECTIONS**

27656 None.

27657 **SEE ALSO**

27658 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <langinfo.h>,
27659 <nl_types.h>

27660 **CHANGE HISTORY**

27661 First released in Issue 2.

27662 **Issue 5**

27663 The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.
27664 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

27665 **NAME**27666 **nrand48** — generate uniformly distributed pseudo-random non-negative long integers27667 **SYNOPSIS**

27668 XSI #include <stdlib.h>

27669 long nrand48(unsigned short xsubi[3]);

27670

27671 **DESCRIPTION**27672 Refer to *drand48()*.

27673 **NAME**27674 `ntohl`, `ntohs` — convert values between host and network byte order27675 **SYNOPSIS**

```
27676     #include <arpa/inet.h>
27677     uint32_t htonl(uint32_t netlong);
27678     uint16_t ntohs(uint16_t netshort);
```

27679 **DESCRIPTION**27680 Refer to `htonl`().

27681 NAME

27682 open — open a file

27683 SYNOPSIS

```
27684 OH #include <sys/stat.h>
27685 #include <fcntl.h>
27686 int open(const char *path, int oflag, ... );
```

27687 DESCRIPTION

The *open()* function shall establish the connection between a file and a file descriptor. It shall create an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The *path* argument points to a pathname naming the file.

The *open()* function shall return a file descriptor for the named file that is the lowest 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 process in the system. The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared.

The file offset used to mark the current position within the file shall be set to the beginning of the file.

The file status flags and file access modes of the open file description shall be set according to the value of *oflag*.

Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in <fcntl.h>. Applications shall specify exactly one of the first three values (file access modes) below in the value of *oflag*:

27703 O_RDONLY Open for reading only.

27704 O_WRONLY Open for writing only.

27705 O_RDWR Open for reading and writing. The result is undefined if this flag is applied to a FIFO.

27707 Any combination of the following may be used:

27708 O_APPEND If set, the file offset shall be set to the end of the file prior to each write.

27709 O_CREAT If the file exists, this flag has no effect except as noted under O_EXCL below. Otherwise, the file shall be created; the user ID of the file shall be set to the effective user ID of the process; the group ID of the file shall be set to the group ID of the file's parent directory or to the effective group ID of the process; and the access permission bits (see <sys/stat.h>) of the file mode shall be set to the value of the third argument taken as type **mode_t** modified as follows: a bitwise AND is performed on the file-mode bits and the corresponding bits in the complement of the process' file mode creation mask. Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared. When bits other than the file permission bits are set, the effect is unspecified. The third argument does not affect whether the file is open for reading, writing, or for both. Implementations shall provide a way to initialize the file's group ID to the group ID of the parent directory. Implementations may, but need not, provide an implementation-defined way to initialize the file's group ID to the effective group ID of the calling process.

27724 SIO_O_DSYNC Write I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion.

27726	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 [EXEXIST], regardless of the contents of the symbolic link. If O_EXCL is set and O_CREAT is not set, the result is undefined.
27733	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.
27735	O_NONBLOCK	When opening a FIFO with O_RDONLY or O_WRONLY set: <ul style="list-style-type: none"> • 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. • 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. When opening a block special or character special file that supports non-blocking opens: <ul style="list-style-type: none"> • 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. • If O_NONBLOCK is clear, the <i>open()</i> function shall block the calling thread until the device is ready or available before returning. Otherwise, the behavior of O_NONBLOCK is unspecified.
27751 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 <i>flags</i> , all I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.
27758 SIO	O_SYNC	Write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.
27760	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.
27765		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.
27768		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.

27770 SIO	If both the O_SYNC and O_DSYNC flags are set, the effect is as if only the O_SYNC flag was set.	
27771		
27772 XSR	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.	
27773		
27774		
27775		
27776		
27777		
27778 XSI	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.	
27779		
27780		
27781	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.	
27782		
27783	RETURN VALUE	
27784	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.	
27785		
27786		
27787	ERRORS	
27788	The <i>open()</i> function shall fail if:	
27789	[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.
27790		
27791		
27792		
27793	[EEXIST]	O_CREAT and O_EXCL are set, and the named file exists.
27794	[EINTR]	A signal was caught during <i>open()</i> .
27795 SIO	[EINVAL]	The implementation does not support synchronized I/O for this file.
27796 XSR	[EIO]	The <i>path</i> argument names a STREAMS file and a hangup or error occurred during the <i>open()</i> .
27797		
27798	[EISDIR]	The named file is a directory and <i>oflag</i> includes O_WRONLY or O_RDWR.
27799	[ELOOP]	A loop exists in symbolic links encountered during resolution of the <i>path</i> argument.
27800		
27801	[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
27802	[ENAMETOOLONG]	The length of the <i>path</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
27803		
27804		
27805	[ENFILE]	The maximum allowable number of files is currently open in the system.
27806	[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.
27807		
27808		
27809 XSR	[ENOSR]	The <i>path</i> argument names a STREAMS-based file and the system is unable to allocate a STREAM.
27810		
27811	[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.
27812		

27813	[ENOTDIR]	A component of the path prefix is not a directory.
27814	[ENXIO]	O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.
27815		
27816	[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
27817		
27818	[EOVERFLOW]	The named file is a regular file and the size of the file cannot be represented correctly in an object of type off_t .
27819		
27820	[EROFS]	The named file resides on a read-only file system and either O_WRONLY, O_RDWR, O_CREAT (if the file does not exist), or O_TRUNC is set in the <i>oflag</i> argument.
27821		
27822		
27823		The <i>open()</i> function may fail if:
27824 XSI	[EAGAIN]	The <i>path</i> argument names the slave side of a pseudo-terminal device that is locked.
27825		
27826	[EINVAL]	The value of the <i>oflag</i> argument is not valid.
27827	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path</i> argument.
27828		
27829	[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}.
27830		
27831		
27832 XSR	[ENOMEM]	The <i>path</i> argument names a STREAMS file and the system is unable to allocate resources.
27833		
27834	[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and <i>oflag</i> is O_WRONLY or O_RDWR.
27835		

27836 EXAMPLES

27837 Opening a File for Writing by the Owner

27838 The following example opens the file **/tmp/file**, either by creating it (if it does not already exist),
 27839 or by truncating its length to 0 (if it does exist). In the former case, if the call creates a new file,
 27840 the access permission bits in the file mode of the file are set to permit reading and writing by the
 27841 owner, and to permit reading only by group members and others.

27842 If the call to *open()* is successful, the file is opened for writing.

```
27843 #include <fcntl.h>
27844 ...
27845 int fd;
27846 mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
27847 char *filename = "/tmp/file";
27848 ...
27849 fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);
27850 ...
```

27851 **Opening a File Using an Existence Check**

27852 The following example uses the `open()` function to try to create the **LOCKFILE** file and open it
27853 for writing. Since the `open()` function specifies the `O_EXCL` flag, the call fails if the file already
27854 exists. In that case, the program assumes that someone else is updating the password file and
27855 exits.

```
27856       #include <fcntl.h>
27857       #include <stdio.h>
27858       #include <stdlib.h>
27859
27860       #define LOCKFILE "/etc/ptmp"
27861       ...
27862       int pfd; /* Integer for file descriptor returned by open() call. */
27863       ...
27864       if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
27865                   S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
27866       {
27867           fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
27868           exit(1);
27869       }
27870       ...
```

27870 **Opening a File for Writing**

27871 The following example opens a file for writing, creating the file if it does not already exist. If the
27872 file does exist, the system truncates the file to zero bytes.

```
27873       #include <fcntl.h>
27874       #include <stdio.h>
27875       #include <stdlib.h>
27876
27877       #define LOCKFILE "/etc/ptmp"
27878       ...
27879       int pfd;
27880       char filename[PATH_MAX+1];
27881       ...
27882       if ((pfd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
27883                   S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
27884       {
27885           perror("Cannot open output file\n");
27886           exit(1);
27887 }
```

27887 **APPLICATION USAGE**

27888 None.

27889 **RATIONALE**

27890 Except as specified in this volume of IEEE Std 1003.1-2001, the flags allowed in *oflag* are not
27891 mutually-exclusive and any number of them may be used simultaneously.

27892 Some implementations permit opening FIFOs with `O_RDWR`. Since FIFOs could be
27893 implemented in other ways, and since two file descriptors can be used to the same effect, this
27894 possibility is left as undefined.

27895 See `getgroups()` about the group of a newly created file.

27896 The use of *open()* to create a regular file is preferable to the use of *creat()*, because the latter is
27897 redundant and included only for historical reasons.

27898 The use of the O_TRUNC flag on FIFOs and directories (pipes cannot be *open()*-ed) must be
27899 permissible without unexpected side effects (for example, *creat()* on a FIFO must not remove
27900 data). Since terminal special files might have type-ahead data stored in the buffer, O_TRUNC
27901 should not affect their content, particularly if a program that normally opens a regular file
27902 should open the current controlling terminal instead. Other file types, particularly
27903 implementation-defined ones, are left implementation-defined.

27904 IEEE Std 1003.1-2001 permits [EACCES] to be returned for conditions other than those explicitly
27905 listed.

27906 The O_NOCTTY flag was added to allow applications to avoid unintentionally acquiring a
27907 controlling terminal as a side effect of opening a terminal file. This volume of
27908 IEEE Std 1003.1-2001 does not specify how a controlling terminal is acquired, but it allows an
27909 implementation to provide this on *open()* if the O_NOCTTY flag is not set and other conditions
27910 specified in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal
27911 Interface are met. The O_NOCTTY flag is an effective no-op if the file being opened is not a
27912 terminal device.

27913 In historical implementations the value of O_RDONLY is zero. Because of that, it is not possible
27914 to detect the presence of O_RDONLY and another option. Future implementations should
27915 encode O_RDONLY and O_WRONLY as bit flags so that:

27916 O_RDONLY | O_WRONLY == O_RDWR

27917 In general, the *open()* function follows the symbolic link if *path* names a symbolic link. However,
27918 the *open()* function, when called with O_CREAT and O_EXCL, is required to fail with [EEXIST]
27919 if *path* names an existing symbolic link, even if the symbolic link refers to a nonexistent file. This
27920 behavior is required so that privileged applications can create a new file in a known location
27921 without the possibility that a symbolic link might cause the file to be created in a different
27922 location.

27923 For example, a privileged application that must create a file with a predictable name in a user-
27924 writable directory, such as the user's home directory, could be compromised if the user creates a
27925 symbolic link with that name that refers to a nonexistent file in a system directory. If the user can
27926 influence the contents of a file, the user could compromise the system by creating a new system
27927 configuration or spool file that would then be interpreted by the system. The test for a symbolic
27928 link which refers to a nonexistent file must be atomic with the creation of a new file.

27929 The POSIX.1-1990 standard required that the group ID of a newly created file be set to the group
27930 ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required
27931 that implementations provide a way to have the group ID be set to the group ID of the
27932 containing directory, but did not prohibit implementations also supporting a way to set the
27933 group ID to the effective group ID of the creating process. Conforming applications should not
27934 assume which group ID will be used. If it matters, an application can use *chown()* to set the
27935 group ID after the file is created, or determine under what conditions the implementation will
27936 set the desired group ID.

27937 FUTURE DIRECTIONS

27938 None.

27939 SEE ALSO

27940 *chmod()*, *close()*, *creat()*, *dup()*, *fcntl()*, *lseek()*, *read()*, *umask()*, *unlockpt()*, *write()*, the Base
27941 Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>, <sys/stat.h>, <sys/types.h>

27942 **CHANGE HISTORY**

27943 First released in Issue 1. Derived from Issue 1 of the SVID.

27944 **Issue 5**27945 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
27946 Threads Extension.

27947 Large File Summit extensions are added.

27948 **Issue 6**

27949 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

27950 The following new requirements on POSIX implementations derive from alignment with the
27951 Single UNIX Specification:

- 27952 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
27953 required for conforming implementations of previous POSIX specifications, it was not
27954 required for UNIX applications.

- 27955 • In the DESCRIPTION, O_CREAT is amended to state that the group ID of the file is set to the
27956 group ID of the file's parent directory or to the effective group ID of the process. This is a
27957 FIPS requirement.

- 27958 • In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file
27959 description. This change is to support large files.

- 27960 • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support
27961 large files.

- 27962 • The [ENXIO] mandatory error condition is added.

- 27963 • The [EINVAL], [ENAMETOOLONG], and [ETXTBSY] optional error conditions are added.

27964 The DESCRIPTION and ERRORS sections are updated so that items related to the optional XSI
27965 STREAMS Option Group are marked.

27966 The following changes were made to align with the IEEE P1003.1a draft standard:

- 27967 • An explanation is added of the effect of the O_CREAT and O_EXCL flags when the path
27968 refers to a symbolic link.

- 27969 • The [ELOOP] optional error condition is added.

27970 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

27971 The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48.

27972 **NAME**

27973 **opendir** — open a directory

27974 **SYNOPSIS**

27975 #include <dirent.h>
27976 DIR *opendir(const char *dirname);

27977 **DESCRIPTION**

27978 The *opendir()* function shall open a directory stream corresponding to the directory named by
27979 the *dirname* argument. The directory stream is positioned at the first entry. If the type **DIR** is
27980 implemented using a file descriptor, applications shall only be able to open up to a total of
27981 {OPEN_MAX} files and directories.

27982 **RETURN VALUE**

27983 Upon successful completion, *opendir()* shall return a pointer to an object of type **DIR**.
27984 Otherwise, a null pointer shall be returned and *errno* set to indicate the error.

27985 **ERRORS**

27986 The *opendir()* function shall fail if:

27987 [EACCES] Search permission is denied for the component of the path prefix of *dirname* or
27988 read permission is denied for *dirname*.

27989 [ELOOP] A loop exists in symbolic links encountered during resolution of the *dirname*
27990 argument.

27991 [ENAMETOOLONG]
27992 The length of the *dirname* argument exceeds {PATH_MAX} or a pathname
27993 component is longer than {NAME_MAX}.

27994 [ENOENT] A component of *dirname* does not name an existing directory or *dirname* is an
27995 empty string.

27996 [ENOTDIR] A component of *dirname* is not a directory.

27997 The *opendir()* function may fail if:

27998 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
27999 resolution of the *dirname* argument.

28000 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

28001 [ENAMETOOLONG]
28002 As a result of encountering a symbolic link in resolution of the *dirname*
28003 argument, the length of the substituted pathname string exceeded
28004 {PATH_MAX}.

28005 [ENFILE] Too many files are currently open in the system.

28006 EXAMPLES

28007 Open a Directory Stream

28008 The following program fragment demonstrates how the *opendir()* function is used.

```
28009 #include <sys/types.h>
28010 #include <dirent.h>
28011 #include <libgen.h>
28012 ...
28013     DIR *dir;
28014     struct dirent *dp;
28015 ...
28016     if ((dir = opendir (".")) == NULL) {
28017         perror ("Cannot open .");
28018         exit (1);
28019     }
28020     while ((dp = readdir (dir)) != NULL) {
28021     ...
28022
```

28022 APPLICATION USAGE

28023 The *opendir()* function should be used in conjunction with *readdir()*, *closedir()*, and *rewinddir()* to
28024 examine the contents of the directory (see the EXAMPLES section in *readdir()*). This method is
28025 recommended for portability.

28026 RATIONALE

28027 Based on historical implementations, the rules about file descriptors apply to directory streams
28028 as well. However, this volume of IEEE Std 1003.1-2001 does not mandate that the directory
28029 stream be implemented using file descriptors. The description of *closedir()* clarifies that if a file
28030 descriptor is used for the directory stream, it is mandatory that *closedir()* deallocate the file
28031 descriptor. When a file descriptor is used to implement the directory stream, it behaves as if the
28032 FD_CLOEXEC had been set for the file descriptor.

28033 The directory entries for dot and dot-dot are optional. This volume of IEEE Std 1003.1-2001 does
28034 not provide a way to test *a priori* for their existence because an application that is portable must
28035 be written to look for (and usually ignore) those entries. Writing code that presumes that they
28036 are the first two entries does not always work, as many implementations permit them to be
28037 other than the first two entries, with a “normal” entry preceding them. There is negligible value
28038 in providing a way to determine what the implementation does because the code to deal with
28039 dot and dot-dot must be written in any case and because such a flag would add to the list of
28040 those flags (which has proven in itself to be objectionable) and might be abused.

28041 Since the structure and buffer allocation, if any, for directory operations are defined by the
28042 implementation, this volume of IEEE Std 1003.1-2001 imposes no portability requirements for
28043 erroneous program constructs, erroneous data, or the use of unspecified values such as the use
28044 or referencing of a *dirp* value or a *dirent* structure value after a directory stream has been closed
28045 or after a *fork()* or one of the *exec* function calls.

28046 FUTURE DIRECTIONS

28047 None.

28048 SEE ALSO

28049 *closedir()*, *Istat()*, *readdir()*, *rewinddir()*, *symlink()*, the Base Definitions volume of
28050 IEEE Std 1003.1-2001, <*dirent.h*>, <*limits.h*>, <*sys/types.h*>

28051 CHANGE HISTORY

28052 First released in Issue 2.

28053 Issue 6

28054 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

28055 The following new requirements on POSIX implementations derive from alignment with the
28056 Single UNIX Specification:

28057 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
28058 required for conforming implementations of previous POSIX specifications, it was not
28059 required for UNIX applications.

28060 • The [ELOOP] mandatory error condition is added.

28061 • A second [ENAMETOOLONG] is added as an optional error condition.

28062 The following changes were made to align with the IEEE P1003.1a draft standard:

28063 • The [ELOOP] optional error condition is added.

28064 **NAME**28065 **openlog** — open a connection to the logging facility28066 **SYNOPSIS**28067 XSI

```
#include <syslog.h>
```

28068

```
void openlog(const char *ident, int logopt, int facility);
```

28069

28070 **DESCRIPTION**28071 Refer to *closelog()*.

28072 NAME

28073 optarg, optarg, optind, optopt — options parsing variables

28074 SYNOPSIS

```
28075 #include <unistd.h>
28076 extern char *optarg;
28077 extern int optarg, optind, optopt;
```

28078 DESCRIPTION

28079 Refer to *getopt()*.

28080 **NAME**
28081 pathconf — get configurable pathname variables

28082 **SYNOPSIS**
28083 #include <unistd.h>
28084 long pathconf(const char **path*, int *name*);

28085 **DESCRIPTION**
28086 Refer to *fpathconf*().

28087 NAME

28088 pause — suspend the thread until a signal is received

28089 SYNOPSIS

```
28090 #include <unistd.h>
28091 int pause(void);
```

28092 DESCRIPTION

28093 The *pause()* function shall suspend the calling thread until delivery of a signal whose action is
28094 either to execute a signal-catching function or to terminate the process.

28095 If the action is to terminate the process, *pause()* shall not return.

28096 If the action is to execute a signal-catching function, *pause()* shall return after the signal-catching
28097 function returns.

28098 RETURN VALUE

28099 Since *pause()* suspends thread execution indefinitely unless interrupted by a signal, there is no
28100 successful completion return value. A value of -1 shall be returned and *errno* set to indicate the
28101 error.

28102 ERRORS

28103 The *pause()* function shall fail if:

28104 [EINTR] A signal is caught by the calling process and control is returned from the
28105 signal-catching function.

28106 EXAMPLES

28107 None.

28108 APPLICATION USAGE

28109 Many common uses of *pause()* have timing windows. The scenario involves checking a
28110 condition related to a signal and, if the signal has not occurred, calling *pause()*. When the signal
28111 occurs between the check and the call to *pause()*, the process often blocks indefinitely. The
28112 *sigprocmask()* and *sigsuspend()* functions can be used to avoid this type of problem.

28113 RATIONALE

28114 None.

28115 FUTURE DIRECTIONS

28116 None.

28117 SEE ALSO

28118 *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

28119 CHANGE HISTORY

28120 First released in Issue 1. Derived from Issue 1 of the SVID.

28121 Issue 5

28122 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

28123 Issue 6

28124 The APPLICATION USAGE section is added.

28125 **NAME**

28126 pclose — close a pipe stream to or from a process

28127 **SYNOPSIS**

28128 CX #include <stdio.h>

28129 int pclose(FILE *stream);

28130

28131 **DESCRIPTION**

28132 The *pclose()* function shall close a stream that was opened by *popen()*, wait for the command to
28133 terminate, and return the termination status of the process that was running the command
28134 language interpreter. However, if a call caused the termination status to be unavailable to
28135 *pclose()*, then *pclose()* shall return -1 with *errno* set to [ECHILD] to report this situation. This can
28136 happen if the application calls one of the following functions:

- 28137 • *wait()*
- 28138 • *waitpid()* with a *pid* argument less than or equal to 0 or equal to the process ID of the
28139 command line interpreter
- 28140 • Any other function not defined in this volume of IEEE Std 1003.1-2001 that could do one of
28141 the above

28142 In any case, *pclose()* shall not return before the child process created by *popen()* has terminated.

28143 If the command language interpreter cannot be executed, the child termination status returned
28144 by *pclose()* shall be as if the command language interpreter terminated using *exit(127)* or
28145 *_exit(127)*.

28146 The *pclose()* function shall not affect the termination status of any child of the calling process
28147 other than the one created by *popen()* for the associated stream.

28148 If the argument *stream* to *pclose()* is not a pointer to a stream created by *popen()*, the result of
28149 *pclose()* is undefined.

28150 **RETURN VALUE**

28151 Upon successful return, *pclose()* shall return the termination status of the command language
28152 interpreter. Otherwise, *pclose()* shall return -1 and set *errno* to indicate the error.

28153 **ERRORS**

28154 The *pclose()* function shall fail if:

28155 [ECHILD] The status of the child process could not be obtained, as described above.

28156 **EXAMPLES**

28157 None.

28158 **APPLICATION USAGE**

28159 None.

28160 **RATIONALE**

28161 There is a requirement that *pclose()* not return before the child process terminates. This is
28162 intended to disallow implementations that return [EINTR] if a signal is received while waiting.
28163 If *pclose()* returned before the child terminated, there would be no way for the application to
28164 discover which child used to be associated with the stream, and it could not do the cleanup
28165 itself.

28166 If the stream pointed to by *stream* was not created by *popen()*, historical implementations of
28167 *pclose()* return -1 without setting *errno*. To avoid requiring *pclose()* to set *errno* in this case,
28168 IEEE Std 1003.1-2001 makes the behavior unspecified. An application should not use *pclose()* to

28169 close any stream that was not created by *popen()*.

28170 Some historical implementations of *pclose()* either block or ignore the signals SIGINT, SIGQUIT,
28171 and SIGHUP while waiting for the child process to terminate. Since this behavior is not
28172 described for the *pclose()* function in IEEE Std 1003.1-2001, such implementations are not
28173 conforming. Also, some historical implementations return [EINTR] if a signal is received, even
28174 though the child process has not terminated. Such implementations are also considered non-
28175 conforming.

28176 Consider, for example, an application that uses:

```
28177     popen( "command" , "r" )
```

28178 to start *command*, which is part of the same application. The parent writes a prompt to its
28179 standard output (presumably the terminal) and then reads from the *popen()*ed stream. The child
28180 reads the response from the user, does some transformation on the response (pathname
28181 expansion, perhaps) and writes the result to its standard output. The parent process reads the
28182 result from the pipe, does something with it, and prints another prompt. The cycle repeats.
28183 Assuming that both processes do appropriate buffer flushing, this would be expected to work.

28184 To conform to IEEE Std 1003.1-2001, *pclose()* must use *waitpid()*, or some similar function,
28185 instead of *wait()*.

28186 The code sample below illustrates how the *pclose()* function might be implemented on a system
28187 conforming to IEEE Std 1003.1-2001.

```
28188     int pclose(FILE *stream)
28189     {
28190         int stat;
28191         pid_t pid;
28192
28193         pid = <pid for process created for stream by popen()>
28194         (void) fclose(stream);
28195         while (waitpid(pid, &stat, 0) == -1) {
28196             if (errno != EINTR){
28197                 stat = -1;
28198                 break;
28199             }
28200         }
28201         return(stat);
28202     }
```

28202 FUTURE DIRECTIONS

28203 None.

28204 SEE ALSO

28205 *fork()*, *popen()*, *waitpid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

28206 CHANGE HISTORY

28207 First released in Issue 1. Derived from Issue 1 of the SVID.

28208 NAME

28209 *perror* — write error messages to standard error

28210 SYNOPSIS

```
28211        #include <stdio.h>
28212        void perror(const char *s);
```

28213 DESCRIPTION

28214 CX The functionality described on this reference page is aligned with the ISO C standard. Any
28215 conflict between the requirements described here and the ISO C standard is unintentional. This
28216 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

28217 The *perror()* function shall map the error number accessed through the symbol *errno* to a
28218 language-dependent error message, which shall be written to the standard error stream as
28219 follows:

- 28220 • First (if *s* is not a null pointer and the character pointed to by *s* is not the null byte), the string
28221 pointed to by *s* followed by a colon and a <space>.
- 28222 • Then an error message string followed by a <newline>.

28223 The contents of the error message strings shall be the same as those returned by *strerror()* with
28224 argument *errno*.

28225 CX The *perror()* function shall mark the file associated with the standard error stream as having
28226 been written (*st_ctime*, *st_mtime* marked for update) at some time between its successful
28227 completion and *exit()*, *abort()*, or the completion of *fflush()* or *fclose()* on *stderr*.

28228 The *perror()* function shall not change the orientation of the standard error stream.

28229 RETURN VALUE

28230 The *perror()* function shall not return a value.

28231 ERRORS

28232 No errors are defined.

28233 EXAMPLES**28234 Printing an Error Message for a Function**

28235 The following example replaces *bufptr* with a buffer that is the necessary size. If an error occurs,
28236 the *perror()* function prints a message and the program exits.

```
28237        #include <stdio.h>
28238        #include <stdlib.h>
28239        ...
28240        char *bufptr;
28241        size_t szbuf;
28242        ...
28243        if ((bufptr = malloc(szbuf)) == NULL) {
28244           perror("malloc");
28245           exit(2);
28246        }
28247        ...
```

28247 APPLICATION USAGE

28248 None.

28249 RATIONALE

28250 None.

28251 FUTURE DIRECTIONS

28252 None.

28253 SEE ALSO

28254 *strerror()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

28255 CHANGE HISTORY

28256 First released in Issue 1. Derived from Issue 1 of the SVID.

28257 Issue 5

28258 A paragraph is added to the DESCRIPTION indicating that *perror()* does not change the orientation of the standard error stream.

28260 Issue 6

28261 Extensions beyond the ISO C standard are marked.

28262 **NAME**

28263 pipe — create an interprocess channel

28264 **SYNOPSIS**

28265 #include <unistd.h>

28266 int pipe(int *fildes*[2]);28267 **DESCRIPTION**

28268 The *pipe()* function shall create a pipe and place two file descriptors, one each into the
28269 arguments *fildes*[0] and *fildes*[1], that refer to the open file descriptions for the read and write
28270 ends of the pipe. Their integer values shall be the two lowest available at the time of the *pipe()*
28271 call. The O_NONBLOCK and FD_CLOEXEC flags shall be clear on both file descriptors. (The
28272 *fcntl()* function can be used to set both these flags.)

28273 Data can be written to the file descriptor *fildes*[1] and read from the file descriptor *fildes*[0]. A
28274 read on the file descriptor *fildes*[0] shall access data written to the file descriptor *fildes*[1] on a
28275 first-in-first-out basis. It is unspecified whether *fildes*[0] is also open for writing and whether
28276 *fildes*[1] is also open for reading.

28277 A process has the pipe open for reading (correspondingly writing) if it has a file descriptor open
28278 that refers to the read end, *fildes*[0] (write end, *fildes*[1]).

28279 Upon successful completion, *pipe()* shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*
28280 fields of the pipe.

28281 **RETURN VALUE**

28282 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
28283 indicate the error.

28284 **ERRORS**

28285 The *pipe()* function shall fail if:

28286 [EMFILE] More than {OPEN_MAX} minus two file descriptors are already in use by this
28287 process.

28288 [ENFILE] The number of simultaneously open files in the system would exceed a
28289 system-imposed limit.

28290 **EXAMPLES**28291 **Using a Pipe to Pass Data Between a Parent Process and a Child Process**

2

28292 The following example demonstrates the use of a pipe to transfer data between a parent process
28293 and a child process. Error handling is excluded, but otherwise this code demonstrates good
28294 practice when using pipes: after the *fork()* the two processes close the unused ends of the pipe
28295 before they commence transferring data.

28296 #include <stdlib.h>

2

28297 #include <unistd.h>

2

28298 ...

2

28299 int *fildes*[2];

2

28300 const int BSIZE = 100;

2

28301 char buf[BSIZE];

2

28302 ssize_t nbytes;

2

28303 int status;

2

28304 status = pipe(*fildes*);

2

28305 if (status == -1) {

2

```
28306     /* an error occurred */
28307     ...
28308 }
28309 switch (fork()) {
28310 case -1: /* Handle error */
28311     break;
28312
28313 case 0: /* Child - reads from pipe */
28314     close(fildes[1]); /* Write end is unused */
28315     nbytes = read(fildes[0], buf, BSIZE); /* Get data from pipe */
28316     /* At this point, a further read would see end of file ... */
28317     close(fildes[0]); /* Finished with pipe */
28318     exit(EXIT_SUCCESS);
28319
28320 default: /* Parent - writes to pipe */
28321     close(fildes[0]); /* Read end is unused */
28322     write(fildes[1], "Hello world\n", 12); /* Write data on pipe */
28323     close(fildes[1]); /* Child will see EOF */
28324     exit(EXIT_SUCCESS);
28325 }
```

28324 APPLICATION USAGE

28325 None.

28326 RATIONALE

28327 The wording carefully avoids using the verb “to open” in order to avoid any implication of use
28328 of *open()*; see also *write()*.

28329 FUTURE DIRECTIONS

28330 None.

28331 SEE ALSO

28332 *fcntl()*, *read()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**,
28333 **<unistd.h>**

28334 CHANGE HISTORY

28335 First released in Issue 1. Derived from Issue 1 of the SVID.

28336 Issue 6

28337 The following new requirements on POSIX implementations derive from alignment with the
28338 Single UNIX Specification:

- The DESCRIPTION is updated to indicate that certain dispositions of *fildes[0]* and *fildes[1]* are unspecified.

28341 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/65 is applied, adding the example to the
28342 EXAMPLES section. 2

28343 NAME

28344 poll — input/output multiplexing

28345 SYNOPSIS

28346 XSI #include <poll.h>

28347 int poll(struct pollfd fds[], nfds_t nfds, int timeout);

28348

28349 DESCRIPTION

28350 The *poll()* function provides applications with a mechanism for multiplexing input/output over
28351 a set of file descriptors. For each member of the array pointed to by *fds*, *poll()* shall examine the
28352 given file descriptor for the event(s) specified in *events*. The number of **pollfd** structures in the
28353 *fds* array is specified by *nfds*. The *poll()* function shall identify those file descriptors on which an
28354 application can read or write data, or on which certain events have occurred.

28355 The *fds* argument specifies the file descriptors to be examined and the events of interest for each
28356 file descriptor. It is a pointer to an array with one member for each open file descriptor of
28357 interest. The array's members are **pollfd** structures within which *fd* specifies an open file
28358 descriptor and *events* and *revents* are bitmasks constructed by OR'ing a combination of the
28359 following event flags:

28360 POLLIN Data other than high-priority data may be read without blocking.

28361 XSR For STREAMS, this flag is set in *revents* even if the message is of zero length.
28362 This flag shall be equivalent to POLLRDNORM | POLLRDBAND.

28363 POLLRDNORM Normal data may be read without blocking.

28364 XSR For STREAMS, data on priority band 0 may be read without blocking. This
28365 flag is set in *revents* even if the message is of zero length.

28366 POLLRDBAND Priority data may be read without blocking.

28367 XSR For STREAMS, data on priority bands greater than 0 may be read without
28368 blocking. This flag is set in *revents* even if the message is of zero length.

28369 POLLPRI High-priority data may be read without blocking.

28370 XSR For STREAMS, this flag is set in *revents* even if the message is of zero length.

28371 POLLOUT Normal data may be written without blocking.

28372 XSR For STREAMS, data on priority band 0 may be written without blocking.

28373 POLLWRNORM Equivalent to POLLOUT.

28374 POLLWRBAND Priority data may be written.

28375 XSR For STREAMS, data on priority bands greater than 0 may be written without
28376 blocking. If any priority band has been written to on this STREAM, this event
28377 only examines bands that have been written to at least once.

28378 POLLERR An error has occurred on the device or stream. This flag is only valid in the
28379 *revents* bitmask; it shall be ignored in the *events* member.

28380 POLLHUP The device has been disconnected. This event and POLLOUT are mutually-
28381 exclusive; a stream can never be writable if a hangup has occurred. However,
28382 this event and POLLIN, POLLRDNORM, POLLRDBAND, or POLLPRI are not
28383 mutually-exclusive. This flag is only valid in the *revents* bitmask; it shall be
28384 ignored in the *events* member.

28385	POLLNVAL	The specified <i>fd</i> value is invalid. This flag is only valid in the <i>revents</i> member; it shall ignored in the <i>events</i> member.
28387		The significance and semantics of normal, priority, and high-priority data are file and device-specific.
28389		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 return from <i>poll()</i> .
28391		In each pollfd structure, <i>poll()</i> shall clear the <i>revents</i> member, except that where the application requested a report on a condition by setting one of the bits of <i>events</i> listed above, <i>poll()</i> shall set the corresponding bit in <i>revents</i> if the requested condition is true. In addition, <i>poll()</i> shall set the POLLHUP, POLLERR, and POLLNVAL flag in <i>revents</i> if the condition is true, even if the application did not set the corresponding bit in <i>events</i> .
28396		If none of the defined events have occurred on any selected file descriptor, <i>poll()</i> shall wait at least <i>timeout</i> milliseconds for an event to occur on any of the selected file descriptors. If the value 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 a requested event occurs or until the call is interrupted.
28400		Implementations may place limitations on the granularity of timeout intervals. If the requested timeout interval requires a finer granularity than the implementation supports, the actual timeout interval shall be rounded up to the next supported value.
28403		The <i>poll()</i> function shall not be affected by the O_NONBLOCK flag.
28404	XSR	The <i>poll()</i> function shall support regular files, terminal and pseudo-terminal devices, FIFOs, pipes, sockets and STREAMS-based files. The behavior of <i>poll()</i> on elements of <i>fds</i> that refer to other types of file is unspecified.
28407		Regular files shall always poll TRUE for reading and writing.
28408		A file descriptor for a socket that is listening for connections shall indicate that it is ready for reading, once connections are available. A file descriptor for a socket that is connecting asynchronously shall indicate that it is ready for writing, once a connection has been established.
28411	RETURN VALUE	
28412		Upon successful completion, <i>poll()</i> shall return a non-negative value. A positive value indicates the total number of file descriptors that have been selected (that is, file descriptors for which the <i>revents</i> member is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, <i>poll()</i> shall return -1 and set <i>errno</i> to indicate the error.
28416	ERRORS	
28417		The <i>poll()</i> function shall fail if:
28418	[EAGAIN]	The allocation of internal data structures failed but a subsequent request may succeed.
28420	[EINTR]	A signal was caught during <i>poll()</i> .
28421	XSR	[EINVAL] The <i>nfds</i> argument is greater than {OPEN_MAX}, or one of the <i>fd</i> members refers to a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.

28424 EXAMPLES

28425 Checking for Events on a Stream

28426 The following example opens a pair of STREAMS devices and then waits for either one to
28427 become writable. This example proceeds as follows:

- 28428 1. Sets the *timeout* parameter to 500 milliseconds.
 - 28429 2. Opens the STREAMS devices **/dev/dev0** and **/dev/dev1**, and then polls them, specifying
28430 POLLOUT and POLLWRBAND as the events of interest.
- 28431 The STREAMS device names **/dev/dev0** and **/dev/dev1** are only examples of how
28432 STREAMS devices can be named; STREAMS naming conventions may vary among
28433 systems conforming to the IEEE Std 1003.1-2001.
- 28434 3. Uses the *ret* variable to determine whether an event has occurred on either of the two
28435 STREAMS. The *poll()* function is given 500 milliseconds to wait for an event to occur (if it
28436 has not occurred prior to the *poll()* call).
 - 28437 4. Checks the returned value of *ret*. If a positive value is returned, one of the following can
28438 be done:
 - 28439 a. Priority data can be written to the open STREAM on priority bands greater than 0,
28440 because the POLLWRBAND event occurred on the open STREAM (*fds[0]* or *fds[1]*).
 - 28441 b. Data can be written to the open STREAM on priority-band 0, because the POLLOUT
28442 event occurred on the open STREAM (*fds[0]* or *fds[1]*).
 - 28443 5. If the returned value is not a positive value, permission to write data to the open STREAM
28444 (on any priority band) is denied.
 - 28445 6. If the POLLHUP event occurs on the open STREAM (*fds[0]* or *fds[1]*), the device on the
28446 open STREAM has disconnected.

```
28447 #include <stropts.h>
28448 #include <poll.h>
28449 ...
28450 struct pollfd fds[2];
28451 int timeout_msecs = 500;
28452 int ret;
28453     int i;

28454 /* Open STREAMS device. */
28455 fds[0].fd = open("/dev/dev0", ...);
28456 fds[1].fd = open("/dev/dev1", ...);
28457 fds[0].events = POLLOUT | POLLWRBAND;
28458 fds[1].events = POLLOUT | POLLWRBAND;
28459
28460     ret = poll(fds, 2, timeout_msecs);
28461
28462     if (ret > 0) {
28463         /* An event on one of the fds has occurred. */
28464         for (i=0; i<2; i++) {
28465             if (fds[i].revents & POLLWRBAND) {
28466                 /* Priority data may be written on device number i. */
28467             ...
28468             }
28469             if (fds[i].revents & POLLOUT) {
```

2
2

```
28468             /* Data may be written on device number i. */
28469     ...
28470     }
28471     if (fds[i].revents & POLLHUP) {
28472         /* A hangup has occurred on device number i. */
28473     ...
28474     }
28475 }
28476 }
```

28477 APPLICATION USAGE

28478 None.

28479 RATIONALE

28480 None.

28481 FUTURE DIRECTIONS

28482 None.

28483 SEE ALSO

28484 Section 2.6 (on page 38), *getmsg()*, *putmsg()*, *read()*, *select()*, *write()*, the Base Definitions volume
28485 of IEEE Std 1003.1-2001, <**poll.h**>, <**stropts.h**>

28486 CHANGE HISTORY

28487 First released in Issue 4, Version 2.

28488 Issue 5

28489 Moved from X/OPEN UNIX extension to BASE.

28490 The description of POLLWRBAND is updated.

28491 Issue 6

28492 Text referring to sockets is added to the DESCRIPTION.

28493 Text relating to the XSI STREAMS Option Group is marked.

28494 The Open Group Corrigendum U055/3 is applied, updating the DESCRIPTION of
28495 POLLWRBAND.

28496 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/66 is applied, correcting the spacing in 2
28497 the EXAMPLES section. 2

28498 NAME

28499 popen — initiate pipe streams to or from a process

28500 SYNOPSIS

28501 CX #include <stdio.h>

28502 FILE *popen(const char *command, const char *mode);

28503

28504 DESCRIPTION

28505 The *popen()* function shall execute the command specified by the string *command*. It shall create a
28506 pipe between the calling program and the executed command, and shall return a pointer to a
28507 stream that can be used to either read from or write to the pipe.

28508 The environment of the executed command shall be as if a child process were created within the
28509 *popen()* call using the *fork()* function, and the child invoked the *sh* utility using the call:

28510 execl(*shell path*, "sh", "-c", *command*, (char *)0);

28511 where *shell path* is an unspecified pathname for the *sh* utility.

28512 The *popen()* function shall ensure that any streams from previous *popen()* calls that remain open
28513 in the parent process are closed in the new child process.

28514 The *mode* argument to *popen()* is a string that specifies I/O mode:

- 28515 1. If *mode* is *r*, when the child process is started, its file descriptor STDOUT_FILENO shall be
28516 the writable end of the pipe, and the file descriptor *fileno(stream)* in the calling process,
28517 where *stream* is the stream pointer returned by *popen()*, shall be the readable end of the
28518 pipe.
- 28519 2. If *mode* is *w*, when the child process is started its file descriptor STDIN_FILENO shall be
28520 the readable end of the pipe, and the file descriptor *fileno(stream)* in the calling process,
28521 where *stream* is the stream pointer returned by *popen()*, shall be the writable end of the
28522 pipe.
- 28523 3. If *mode* is any other value, the result is undefined.

28524 After *popen()*, both the parent and the child process shall be capable of executing independently
28525 before either terminates.

28526 Pipe streams are byte-oriented.

28527 RETURN VALUE

28528 Upon successful completion, *popen()* shall return a pointer to an open stream that can be used to
28529 read or write to the pipe. Otherwise, it shall return a null pointer and may set *errno* to indicate
28530 the error.

28531 ERRORS

28532 The *popen()* function may fail if:

28533 [EMFILE] {FOPEN_MAX} or {STREAM_MAX} streams are currently open in the calling
28534 process.

28535 [EINVAL] The *mode* argument is invalid.

28536 The *popen()* function may also set *errno* values as described by *fork()* or *pipe()*.

28537 EXAMPLES

28538	Using popen() to Obtain a List of Files from the ls Utility	2
28539	The following example demonstrates the use of <i>popen()</i> and <i>pclose()</i> to execute the command <i>ls *</i>	2
28540	in order to obtain a list of files in the current directory:	2
28541	#include <stdio.h>	2
28542	...	2
28543	FILE *fp;	2
28544	int status;	2
28545	char path[PATH_MAX];	2
28546	fp = popen("ls *", "r");	2
28547	if (fp == NULL)	2
28548	/* Handle error */	2
28549	while (fgets(path, PATH_MAX, fp) != NULL)	2
28550	printf("%s", path);	2
28551	status = pclose(fp);	2
28552	if (status == -1) {	2
28553	/* Error reported by pclose() */	2
28554	...	2
28555	} else {	2
28556	/* Use macros described under wait() to inspect 'status' in order	2
28557	to determine success/failure of command executed by popen() */	2
28558	...	2
28559	}	2

28560 APPLICATION USAGE

28561 Since open files are shared, a mode *r* command can be used as an input filter and a mode *w* command as an output filter.

28563 Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be prevented by careful buffer flushing; for example, with *fflush()*.

28566 A stream opened by *popen()* should be closed by *pclose()*.

28567 The behavior of *popen()* is specified for values of *mode* of *r* and *w*. Other modes such as *rb* and *wb* might be supported by specific implementations, but these would not be portable features. Note that historical implementations of *popen()* only check to see if the first character of *mode* is *r*. Thus, a *mode* of *robert the robot* would be treated as *mode r*, and a *mode* of *anything else* would be treated as *mode w*.

28572 If the application calls *waitpid()* or *waitid()* with a *pid* argument greater than 0, and it still has a stream that was called with *popen()* open, it must ensure that *pid* does not refer to the process started by *popen()*.

28575 To determine whether or not the environment specified in the Shell and Utilities volume of IEEE Std 1003.1-2001 is present, use the function call:

28577 *sysconf(_SC_2_VERSION)*

28578 (See *sysconf()*).

28579 RATIONALE

The `popen()` function should not be used by programs that have set user (or group) ID privileges. The `fork()` and `exec` family of functions (except `execvp()`), should be used instead. This prevents any unforeseen manipulation of the environment of the user that could cause execution of commands not anticipated by the calling program.

If the original and *popen()*ed processes both intend to read or write or read and write a common file, and either will be using FILE-type C functions (*fread()*, *fwrite()*, and so on), the rules for sharing file handles must be observed (see Section 2.5.1 (on page 35)).

28587 FUTURE DIRECTIONS

28588 None.

28589 SEE ALSO

28590 *pclose(), pipe(), sysconf(), system()*, the Base Definitions volume of IEEE Std 1003.1-2001,
28591 *<stdio.h>*, the Shell and Utilities volume of IEEE Std 1003.1-2001, *sh*

28592 CHANGE HISTORY

28593 First released in Issue 1. Derived from Issue 1 of the SVID.

28594 Issue 5

28595 A statement is added to the DESCRIPTION indicating that pipe streams are byte-oriented.

28596 Issue 6

28597 The following new requirements on POSIX implementations derive from alignment with the
28598 Single UNIX Specification:

- The optional [EMFILE] error condition is added.

28600 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/67 is applied, adding the example to the 28601 EXAMPLES section. 2

28602 NAME

28603 posix_fadvise — file advisory information (**ADVANCED REALTIME**)

28604 SYNOPSIS

28605 ADV #include <fcntl.h>

28606 int posix_fadvise(int *fd*, off_t *offset*, off_t *len*, int *advice*);

28607

2

28608 DESCRIPTION

28609 The *posix_fadvise()* function shall advise the implementation on the expected behavior of the
28610 application with respect to the data in the file associated with the open file descriptor, *fd*,
28611 starting at *offset* and continuing for *len* bytes. The specified range need not currently exist in the
28612 file. If *len* is zero, all data following *offset* is specified. The implementation may use this
28613 information to optimize handling of the specified data. The *posix_fadvise()* function shall have no
28614 effect on the semantics of other operations on the specified data, although it may affect the
28615 performance of other operations.

28616 The advice to be applied to the data is specified by the *advice* parameter and may be one of the
28617 following values:

28618 POSIX_FADV_NORMAL

28619 Specifies that the application has no advice to give on its behavior with respect to the
28620 specified data. It is the default characteristic if no advice is given for an open file.

28621 POSIX_FADV_SEQUENTIAL

28622 Specifies that the application expects to access the specified data sequentially from lower
28623 offsets to higher offsets.

28624 POSIX_FADV_RANDOM

28625 Specifies that the application expects to access the specified data in a random order.

28626 POSIX_FADV_WILLNEED

28627 Specifies that the application expects to access the specified data in the near future.

28628 POSIX_FADV_DONTNEED

28629 Specifies that the application expects that it will not access the specified data in the near
28630 future.

28631 POSIX_FADV_NOREUSE

28632 Specifies that the application expects to access the specified data once and then not reuse it
28633 thereafter.

28634 These values are defined in <fcntl.h>.

28635 RETURN VALUE

28636 Upon successful completion, *posix_fadvise()* shall return zero; otherwise, an error number shall
28637 be returned to indicate the error.

28638 ERRORS

28639 The *posix_fadvise()* function shall fail if:

28640 [EBADF] The *fd* argument is not a valid file descriptor.

28641 [EINVAL] The value of *advice* is invalid.

28642 [ESPIPE] The *fd* argument is associated with a pipe or FIFO.

28643 EXAMPLES

28644 None.

28645 APPLICATION USAGE

28646 The *posix_fadvise()* function is part of the Advisory Information option and need not be provided
28647 on all implementations.

28648 RATIONALE

28649 None.

28650 FUTURE DIRECTIONS

28651 None.

28652 SEE ALSO

28653 *posix_madvise()*, the Base Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>

28654 CHANGE HISTORY

28655 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28656 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

28657 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/68 is applied, changing the function 2
28658 prototype in the SYNOPSIS section. The previous prototype was not large file-aware, and the 2
28659 standard developers felt it acceptable to make this change before implementations of this 2
28660 function become widespread. 2

28661 NAME

28662 posix_fallocate — file space control (**ADVANCED REALTIME**)

28663 SYNOPSIS

28664 ADV #include <fcntl.h>

```
28665      int posix_fallocate(int fd, off_t offset, off_t len);
```

28666

2

28667 DESCRIPTION

28668 The *posix_fallocate()* function shall ensure that any required storage for regular file data starting
28669 at *offset* and continuing for *len* bytes is allocated on the file system storage media. If
28670 *posix_fallocate()* returns successfully, subsequent writes to the specified file data shall not fail
28671 due to the lack of free space on the file system storage media.

28672 If the *offset+len* is beyond the current file size, then *posix_fallocate()* shall adjust the file size to
28673 *offset+len*. Otherwise, the file size shall not be changed.

28674 It is implementation-defined whether a previous *posix_fadvise()* call influences allocation
28675 strategy.

28676 Space allocated via *posix_fallocate()* shall be freed by a successful call to *creat()* or *open()* that
28677 truncates the size of the file. Space allocated via *posix_fallocate()* may be freed by a successful call
28678 to *ftruncate()* that reduces the file size to a size smaller than *offset+len*.

28679 RETURN VALUE

28680 Upon successful completion, *posix_fallocate()* shall return zero; otherwise, an error number shall
28681 be returned to indicate the error.

28682 ERRORS

28683 The *posix_fallocate()* function shall fail if:

- 28684 [EBADF] The *fd* argument is not a valid file descriptor.
- 28685 [EBADF] The *fd* argument references a file that was opened without write permission.
- 28686 [EFBIG] The value of *offset+len* is greater than the maximum file size.
- 28687 [EINTR] A signal was caught during execution.
- 28688 [EINVAL] The *len* argument was zero or the *offset* argument was less than zero.
- 28689 [EIO] An I/O error occurred while reading from or writing to a file system.
- 28690 [ENODEV] The *fd* argument does not refer to a regular file.
- 28691 [ENOSPC] There is insufficient free space remaining on the file system storage media.
- 28692 [ESPIPE] The *fd* argument is associated with a pipe or FIFO.

28693 EXAMPLES

28694 None.

28695 APPLICATION USAGE

28696 The *posix_fallocate()* function is part of the Advisory Information option and need not be
28697 provided on all implementations.

28698 RATIONALE

28699 None.

28700 FUTURE DIRECTIONS

28701 None.

28702 SEE ALSO

28703 *creat()*, *ftruncate()*, *open()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001,
28704 *<fcntl.h>*

28705 CHANGE HISTORY

28706 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28707 In the SYNOPSIS, the inclusion of *<sys/types.h>* is no longer required.

28708 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/69 is applied, changing the function 2
28709 prototype in the SYNOPSIS section. The previous prototype was not large file-aware, and the 2
28710 standard developers felt it acceptable to make this change before implementations of this 2
28711 function become widespread. 2

28712 NAME

28713 posix_madvise — memory advisory information and alignment control (ADVANCED
28714 REALTIME)

28715 SYNOPSIS

28716 ADV #include <sys/mman.h>
28717 int posix_madvise(void *addr, size_t len, int advice);
28718

28719 DESCRIPTION

28720 MF|SHM The *posix_madvise()* function need only be supported if either the Memory Mapped Files or the Shared Memory Objects options are supported.

28722 The *posix_madvise()* function shall advise the implementation on the expected behavior of the application with respect to the data in the memory starting at address *addr*, and continuing for *len* bytes. The implementation may use this information to optimize handling of the specified data. The *posix_madvise()* function shall have no effect on the semantics of access to memory in the specified range, although it may affect the performance of access.

28727 The implementation may require that *addr* be a multiple of the page size, which is the value returned by *sysconf()* when the name value *_SC_PAGESIZE* is used.

28729 The advice to be applied to the memory range is specified by the *advice* parameter and may be one of the following values:

28731 POSIX_MADV_NORMAL

28732 Specifies that the application has no advice to give on its behavior with respect to the specified range. It is the default characteristic if no advice is given for a range of memory.

28734 POSIX_MADV_SEQUENTIAL

28735 Specifies that the application expects to access the specified range sequentially from lower addresses to higher addresses.

28737 POSIX_MADV_RANDOM

28738 Specifies that the application expects to access the specified range in a random order.

28739 POSIX_MADV_WILLNEED

28740 Specifies that the application expects to access the specified range in the near future.

28741 POSIX_MADV_DONTNEED

28742 Specifies that the application expects that it will not access the specified range in the near future.

28744 These values are defined in the <sys/mman.h> header.

28745 RETURN VALUE

28746 Upon successful completion, *posix_madvise()* shall return zero; otherwise, an error number shall
28747 be returned to indicate the error.

28748 ERRORS

28749 The *posix_madvise()* function shall fail if:

28750 [EINVAL] The value of *advice* is invalid.

28751 [ENOMEM] Addresses in the range starting at *addr* and continuing for *len* bytes are partly
28752 or completely outside the range allowed for the address space of the calling
28753 process.

28754 The *posix_madvise()* function may fail if:
28755 [EINVAL] The value of *addr* is not a multiple of the value returned by *sysconf()* when the
28756 name value *_SC_PAGESIZE* is used.
28757 [EINVAL] The value of *len* is zero.

28758 EXAMPLES

28759 None.

28760 APPLICATION USAGE

28761 The *posix_madvise()* function is part of the Advisory Information option and need not be
28762 provided on all implementations.

28763 RATIONALE

28764 None.

28765 FUTURE DIRECTIONS

28766 None.

28767 SEE ALSO

28768 *mmap()*, *posix_fadvise()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001,
28769 <sys/mman.h>

28770 CHANGE HISTORY

28771 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28772 IEEE PASC Interpretation 1003.1 #102 is applied.

28773 NAME

28774 posix_mem_offset — find offset and length of a mapped typed memory block (**ADVANCED**
28775 **REALTIME**)

28776 SYNOPSIS

28777 TTY #include <sys/mman.h>

28778 int posix_mem_offset(const void *restrict addr, size_t len,
28779 off_t *restrict off, size_t *restrict contig_len,
28780 int *restrict fildes);

28781

28782 DESCRIPTION

28783 The *posix_mem_offset()* function shall return in the variable pointed to by *off* a value that
28784 identifies the offset (or location), within a memory object, of the memory block currently
28785 mapped at *addr*. The function shall return in the variable pointed to by *fildes*, the descriptor used
28786 (via *mmap()*) to establish the mapping which contains *addr*. If that descriptor was closed since
28787 the mapping was established, the returned value of *fildes* shall be -1. The *len* argument specifies
28788 the length of the block of the memory object the user wishes the offset for; upon return, the value
28789 pointed to by *contig_len* shall equal either *len*, or the length of the largest contiguous block of the
28790 memory object that is currently mapped to the calling process starting at *addr*, whichever is
28791 smaller.

28792 If the memory object mapped at *addr* is a typed memory object, then if the *off* and *contig_len*
28793 values obtained by calling *posix_mem_offset()* are used in a call to *mmap()* with a file descriptor
28794 that refers to the same memory pool as *fildes* (either through the same port or through a different
28795 port), and that was opened with neither the **POSIX_TYPED_MEM_ALLOCATE** nor the
28796 **POSIX_TYPED_MEM_ALLOCATE_CONTIG** flag, the typed memory area that is mapped shall
28797 be exactly the same area that was mapped at *addr* in the address space of the process that called
28798 *posix_mem_offset()*.

28799 If the memory object specified by *fildes* is not a typed memory object, then the behavior of this
28800 function is implementation-defined.

28801 RETURN VALUE

28802 Upon successful completion, the *posix_mem_offset()* function shall return zero; otherwise, the
28803 corresponding error status value shall be returned.

28804 ERRORS

28805 The *posix_mem_offset()* function shall fail if:

28806 [EACCES] The process has not mapped a memory object supported by this function at
28807 the given address *addr*.

28808 This function shall not return an error code of [EINTR].

28809 EXAMPLES

28810 None.

28811 APPLICATION USAGE

28812 None.

28813 RATIONALE

28814 None.

28815 **FUTURE DIRECTIONS**

28816 None.

28817 **SEE ALSO**

28818 *mmap()*, *posix_typed_mem_open()*, the Base Definitions volume of IEEE Std 1003.1-2001,
28819 <sys/mman.h>

28820 **CHANGE HISTORY**

28821 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

28822 NAME

28823 posix_memalign — aligned memory allocation (**ADVANCED REALTIME**)

28824 SYNOPSIS

28825 ADV #include <stdlib.h>

```
28826        int posix_memalign(void **memptr, size_t alignment, size_t size);
```

28827

28828 DESCRIPTION

28829 The *posix_memalign()* function shall allocate *size* bytes aligned on a boundary specified by
28830 *alignment*, and shall return a pointer to the allocated memory in *memptr*. The value of *alignment*
28831 shall be a multiple of *sizeof(void *)*, that is also a power of two. Upon successful completion, the
28832 value pointed to by *memptr* shall be a multiple of *alignment*.

28833 CX The *free()* function shall deallocate memory that has previously been allocated by
28834 *posix_memalign()*.

28835 RETURN VALUE

28836 Upon successful completion, *posix_memalign()* shall return zero; otherwise, an error number
28837 shall be returned to indicate the error.

28838 ERRORS

28839 The *posix_memalign()* function shall fail if:

28840 [EINVAL] The value of the alignment parameter is not a power of two multiple of
28841 *sizeof(void *)*.

28842 [ENOMEM] There is insufficient memory available with the requested alignment.

28843 EXAMPLES

28844 None.

28845 APPLICATION USAGE

28846 The *posix_memalign()* function is part of the Advisory Information option and need not be
28847 provided on all implementations.

28848 RATIONALE

28849 None.

28850 FUTURE DIRECTIONS

28851 None.

28852 SEE ALSO

28853 *free()*, *malloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

28854 CHANGE HISTORY

28855 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

28856 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

28857 NAME

28858 **posix_openpt** — open a pseudo-terminal device

28859 SYNOPSIS

```
28860 XSI #include <stdlib.h>
28861 #include <fcntl.h>
28862 int posix_openpt(int oflag);
28863
```

28864 DESCRIPTION

28865 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.

28868 The file status flags and file access modes of the open file description shall be set according to
28869 the value of *oflag*.

28870 Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list,
28871 defined in <fcntl.h>:

28872 O_RDWR Open for reading and writing.

28873 O_NOCTTY If set *posix_openpt()* shall not cause the terminal device to become the
28874 controlling terminal for the process.

28875 The behavior of other values for the *oflag* argument is unspecified.

28876 RETURN VALUE

28877 Upon successful completion, the *posix_openpt()* function shall open a master pseudo-terminal
28878 device and return a non-negative integer representing the lowest numbered unused file
28879 descriptor. Otherwise, -1 shall be returned and *errno* set to indicate the error.

28880 ERRORS

28881 The *posix_openpt()* function shall fail if:

28882 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

28883 [ENFILE] The maximum allowable number of files is currently open in the system.

28884 The *posix_openpt()* function may fail if:

28885 [EINVAL] The value of *oflag* is not valid.

28886 [EAGAIN] Out of pseudo-terminal resources.

28887 XSR [ENOSR] Out of STREAMS resources.

28888 EXAMPLES**28889 Opening a Pseudo-Terminal and Returning the Name of the Slave Device and a File
28890 Descriptor**

```
28891 #include <fcntl.h>
28892 #include <stdio.h>
28893
28894 int masterfd, slavefd;
28895 char *slavedevice;
28896
28897 masterfd = posix_openpt(O_RDWR | O_NOCTTY);
28898 if (masterfd == -1
28899     || grantpt (masterfd) == -1
```

```
28898     || unlockpt (masterfd) == -1
28899     || (slavedevice = ptsname (masterfd)) == NULL)
28900     return -1;
28901
28902     printf("slave device is: %s\n", slavedevice);
28903     slavefd = open(slave, O_RDWR|O_NOCTTY);
28904     if (slavefd < 0)
28905         return -1;
```

28905 APPLICATION USAGE

28906 This function is a method for portably obtaining a file descriptor of a master terminal device for
28907 a pseudo-terminal. The *grantpt()* and *ptsname()* functions can be used to manipulate mode and
28908 ownership permissions, and to obtain the name of the slave device, respectively.

28909 RATIONALE

28910 The standard developers considered the matter of adding a special device for cloning master
28911 pseudo-terminals: the **/dev/ptmx** device. However, consensus could not be reached, and it was
28912 felt that adding a new function would permit other implementations. The *posix_openpt()*
28913 function is designed to complement the *grantpt()*, *ptsname()*, and *unlockpt()* functions.

28914 On implementations supporting the **/dev/ptmx** clone device, opening the master device of a
28915 pseudo-terminal is simply:

```
28916 mfdp = open( "/dev/ptmx" , oflag );
28917 if (mfdp < 0)
28918     return -1;
```

28919 FUTURE DIRECTIONS

28920 None.

28921 SEE ALSO

28922 *grantpt()*, *open()*, *ptsname()*, *unlockpt()*, the Base Definitions volume of IEEE Std 1003.1-2001,
28923 **<fcntl.h>**

28924 CHANGE HISTORY

28925 First released in Issue 6.

28926 NAME

28927 posix_spawn, posix_spawnp — spawn a process (**ADVANCED REALTIME**)

28928 SYNOPSIS

28929 SPN #include <spawn.h>

```

28930     int posix_spawn(pid_t *restrict pid, const char *restrict path,
28931             const posix_spawn_file_actions_t *file_actions,
28932             const posix_spawnattr_t *restrict attrp,
28933             char *const argv[restrict], char *const envp[restrict]);
28934     int posix_spawnp(pid_t *restrict pid, const char *restrict file,
28935             const posix_spawn_file_actions_t *file_actions,
28936             const posix_spawnattr_t *restrict attrp,
28937             char *const argv[restrict], char * const envp[restrict]);
28938

```

28939 DESCRIPTION

28940 The *posix_spawn()* and *posix_spawnp()* functions shall create a new process (child process) from
 28941 the specified process image. The new process image shall be constructed from a regular
 28942 executable file called the new process image file.

28943 When a C program is executed as the result of this call, it shall be entered as a C-language
 28944 function call as follows:

```
28945     int main(int argc, char *argv[]);
```

28946 where *argc* is the argument count and *argv* is an array of character pointers to the arguments
 28947 themselves. In addition, the following variable:

```
28948     extern char **environ;
```

28949 shall be initialized as a pointer to an array of character pointers to the environment strings.

28950 The argument *argv* is an array of character pointers to null-terminated strings. The last member
 28951 of this array shall be a null pointer and is not counted in *argc*. These strings constitute the
 28952 argument list available to the new process image. The value in *argv[0]* should point to a filename
 28953 that is associated with the process image being started by the *posix_spawn()* or *posix_spawnp()*
 28954 function.

28955 The argument *envp* is an array of character pointers to null-terminated strings. These strings
 28956 constitute the environment for the new process image. The environment array is terminated by a
 28957 null pointer.

28958 The number of bytes available for the child process' combined argument and environment lists
 28959 is {ARG_MAX}. The implementation shall specify in the system documentation (see the Base
 28960 Definitions volume of IEEE Std 1003.1-2001, Chapter 2, Conformance) whether any list
 28961 overhead, such as length words, null terminators, pointers, or alignment bytes, is included in
 28962 this total.

28963 The *path* argument to *posix_spawn()* is a pathname that identifies the new process image file to
 28964 execute.

28965 The *file* parameter to *posix_spawnp()* shall be used to construct a pathname that identifies the
 28966 new process image file. If the *file* parameter contains a slash character, the *file* parameter shall be
 28967 used as the pathname for the new process image file. Otherwise, the path prefix for this file shall
 28968 be obtained by a search of the directories passed as the environment variable *PATH* (see the Base
 28969 Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables). If this
 28970 environment variable is not defined, the results of the search are implementation-defined.

If *file_actions* is a null pointer, then file descriptors open in the calling process shall remain open in the child process, except for those whose close-on-exec flag FD_CLOEXEC is set (see *fctl()*). For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks (see *fctl()*), shall remain unchanged.

If *file_actions* is not NULL, then the file descriptors open in the child process shall be those open in the calling process as modified by the spawn file actions object pointed to by *file_actions* and the FD_CLOEXEC flag of each remaining open file descriptor after the spawn file actions have been processed. The effective order of processing the spawn file actions shall be:

1. The set of open file descriptors for the child process shall initially be the same set as is open for the calling process. All attributes of the corresponding open file descriptions, including file locks (see *fctl()*), shall remain unchanged.
2. The signal mask, signal default actions, and the effective user and group IDs for the child process shall be changed as specified in the attributes object referenced by *attrp*.
3. The file actions specified by the spawn file actions object shall be performed in the order in which they were added to the spawn file actions object.
4. Any file descriptor that has its FD_CLOEXEC flag set (see *fctl()*) shall be closed.

The **posix_spawnattr_t** spawn attributes object type is defined in <**spawn.h**>. It shall contain at least the attributes defined below.

If the POSIX_SPAWN_SETPGROUP flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is non-zero, then the child's process group shall be as specified in the *spawn-pgroup* attribute of the object referenced by *attrp*.

As a special case, if the POSIX_SPAWN_SETPGROUP flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is set to zero, then the child shall be in a new process group with a process group ID equal to its process ID.

If the POSIX_SPAWN_SETPGROUP flag is not set in the *spawn-flags* attribute of the object referenced by *attrp*, the new child process shall inherit the parent's process group.

If the POSIX_SPAWN_SETSCHEDPARAM flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, but POSIX_SPAWN_SETSCHEDULER is not set, the new process image shall initially have the scheduling policy of the calling process with the scheduling parameters specified in the *spawn-schedparam* attribute of the object referenced by *attrp*.

If the POSIX_SPAWN_SETSCHEDULER flag is set in the *spawn-flags* attribute of the object referenced by *attrp* (regardless of the setting of the POSIX_SPAWN_SETSCHEDPARAM flag), the new process image shall initially have the scheduling policy specified in the *spawn-schedpolicy* attribute of the object referenced by *attrp* and the scheduling parameters specified in the *spawn-schedparam* attribute of the same object.

The POSIX_SPAWN_RESETIDS flag in the *spawn-flags* attribute of the object referenced by *attrp* governs the effective user ID of the child process. If this flag is not set, the child process shall inherit the parent process' effective user ID. If this flag is set, the child process' effective user ID shall be reset to the parent's real user ID. In either case, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the child process shall become that file's owner ID before the new process image begins execution.

The POSIX_SPAWN_RESETIDS flag in the *spawn-flags* attribute of the object referenced by *attrp* also governs the effective group ID of the child process. If this flag is not set, the child process shall inherit the parent process' effective group ID. If this flag is set, the child process' effective group ID shall be reset to the parent's real group ID. In either case, if the set-group-ID mode bit

29017 of the new process image file is set, the effective group ID of the child process shall become that
29018 file's group ID before the new process image begins execution.

29019 If the `POSIX_SPAWN_SETSIGMASK` flag is set in the *spawn-flags* attribute of the object
29020 referenced by *attrp*, the child process shall initially have the signal mask specified in the *spawn-*
29021 *sigmask* attribute of the object referenced by *attrp*.

29022 If the `POSIX_SPAWN_SETSIGDEF` flag is set in the *spawn-flags* attribute of the object referenced
29023 by *attrp*, the signals specified in the *spawn-sigdefault* attribute of the same object shall be set to
29024 their default actions in the child process. Signals set to the default action in the parent process
29025 shall be set to the default action in the child process.

29026 Signals set to be caught by the calling process shall be set to the default action in the child
29027 process.

29028 Except for `SIGCHLD`, signals set to be ignored by the calling process image shall be set to be
29029 ignored by the child process, unless otherwise specified by the `POSIX_SPAWN_SETSIGDEF` flag
29030 being set in the *spawn-flags* attribute of the object referenced by *attrp* and the signals being
29031 indicated in the *spawn-sigdefault* attribute of the object referenced by *attrp*.

29032 If the `SIGCHLD` signal is set to be ignored by the calling process, it is unspecified whether the
29033 `SIGCHLD` signal is set to be ignored or to the default action in the child process, unless
29034 otherwise specified by the `POSIX_SPAWN_SETSIGDEF` flag being set in the *spawn_flags*
29035 attribute of the object referenced by *attrp* and the `SIGCHLD` signal being indicated in the
29036 *spawn_sigdefault* attribute of the object referenced by *attrp*.

29037 If the value of the *attrp* pointer is `NULL`, then the default values are used.

29038 All process attributes, other than those influenced by the attributes set in the object referenced
29039 by *attrp* as specified above or by the file descriptor manipulations specified in *file_actions*, shall
29040 appear in the new process image as though `fork()` had been called to create a child process and
29041 then a member of the `exec` family of functions had been called by the child process to execute the
29042 new process image.

29043 THR It is implementation-defined whether the fork handlers are run when `posix_spawn()` or
29044 `posix_spawnp()` is called.

29045 RETURN VALUE

29046 Upon successful completion, `posix_spawn()` and `posix_spawnp()` shall return the process ID of the
29047 child process to the parent process, in the variable pointed to by a non-`NULL` *pid* argument, and
29048 shall return zero as the function return value. Otherwise, no child process shall be created, the
29049 value stored into the variable pointed to by a non-`NULL` *pid* is unspecified, and an error number
29050 shall be returned as the function return value to indicate the error. If the *pid* argument is a null
29051 pointer, the process ID of the child is not returned to the caller.

29052 ERRORS

29053 The `posix_spawn()` and `posix_spawnp()` functions may fail if:

29054 [EINVAL] The value specified by *file_actions* or *attrp* is invalid.

29055 If this error occurs after the calling process successfully returns from the `posix_spawn()` or
29056 `posix_spawnp()` function, the child process may exit with exit status 127.

29057 If `posix_spawn()` or `posix_spawnp()` fail for any of the reasons that would cause `fork()` or one of
29058 the `exec` family of functions to fail, an error value shall be returned as described by `fork()` and
29059 `exec`, respectively (or, if the error occurs after the calling process successfully returns, the child
29060 process shall exit with exit status 127).

29061 If POSIX_SPAWN_SETPGROUP is set in the *spawn-flags* attribute of the object referenced by *attrp*, and *posix_spawn()* or *posix_spawnp()* fails while changing the child's process group, an
29062 error value shall be returned as described by *setpgid()* (or, if the error occurs after the calling
29063 process successfully returns, the child process shall exit with exit status 127).
29064

29065 PS If POSIX_SPAWN_SETSCHEDPARAM is set and POSIX_SPAWN_SETSCHEDULER is not set
29066 in the *spawn-flags* attribute of the object referenced by *attrp*, then if *posix_spawn()* or
29067 *posix_spawnp()* fails for any of the reasons that would cause *sched_setparam()* to fail, an error
29068 value shall be returned as described by *sched_setparam()* (or, if the error occurs after the calling
29069 process successfully returns, the child process shall exit with exit status 127).
29070

29071 If POSIX_SPAWN_SETSCHEDULER is set in the *spawn-flags* attribute of the object referenced by
29072 *attrp*, and if *posix_spawn()* or *posix_spawnp()* fails for any of the reasons that would cause
29073 *sched_setscheduler()* to fail, an error value shall be returned as described by *sched_setscheduler()*
29074 (or, if the error occurs after the calling process successfully returns, the child process shall exit
with exit status 127).
29075

29076 If the *file_actions* argument is not NULL, and specifies any *close*, *dup2*, or *open* actions to be
29077 performed, and if *posix_spawn()* or *posix_spawnp()* fails for any of the reasons that would cause
29078 *close()*, *dup2()*, or *open()* to fail, an error value shall be returned as described by *close()*, *dup2()*,
29079 and *open()*, respectively (or, if the error occurs after the calling process successfully returns, the
29080 child process shall exit with exit status 127). An open file action may, by itself, result in any of
the errors described by *close()* or *dup2()*, in addition to those described by *open()*.

29081 EXAMPLES

29082 None.

29083 APPLICATION USAGE

29084 These functions are part of the Spawn option and need not be provided on all implementations.

29085 RATIONALE

29086 The *posix_spawn()* function and its close relation *posix_spawnp()* have been introduced to
29087 overcome the following perceived difficulties with *fork()*: the *fork()* function is difficult or
29088 impossible to implement without swapping or dynamic address translation.

- 29089 • Swapping is generally too slow for a realtime environment.
- 29090 • Dynamic address translation is not available everywhere that POSIX might be useful.
- 29091 • Processes are too useful to simply option out of POSIX whenever it must run without
29092 address translation or other MMU services.

29093 Thus, POSIX needs process creation and file execution primitives that can be efficiently
29094 implemented without address translation or other MMU services.

29095 The *posix_spawn()* function is implementable as a library routine, but both *posix_spawn()* and
29096 *posix_spawnp()* are designed as kernel operations. Also, although they may be an efficient
29097 replacement for many *fork()*/*exec* pairs, their goal is to provide useful process creation
29098 primitives for systems that have difficulty with *fork()*, not to provide drop-in replacements for
29099 *fork()*/*exec*.

29100 This view of the role of *posix_spawn()* and *posix_spawnp()* influenced the design of their API. It
29101 does not attempt to provide the full functionality of *fork()*/*exec* in which arbitrary user-specified
29102 operations of any sort are permitted between the creation of the child process and the execution
29103 of the new process image; any attempt to reach that level would need to provide a programming
29104 language as parameters. Instead, *posix_spawn()* and *posix_spawnp()* are process creation
29105 primitives like the *Start_Process* and *Start_Process_Search* Ada language bindings package
29106 *POSIX_Process_Primitives* and also like those in many operating systems that are not UNIX

29107 systems, but with some POSIX-specific additions.

29108 To achieve its coverage goals, *posix_spawn()* and *posix_spawnp()* have control of six types of
29109 inheritance: file descriptors, process group ID, user and group ID, signal mask, scheduling, and
29110 whether each signal ignored in the parent will remain ignored in the child, or be reset to its
29111 default action in the child.

29112 Control of file descriptors is required to allow an independently written child process image to
29113 access data streams opened by and even generated or read by the parent process without being
29114 specifically coded to know which parent files and file descriptors are to be used. Control of the
29115 process group ID is required to control how the child process' job control relates to that of the
29116 parent.

29117 Control of the signal mask and signal defaulting is sufficient to support the implementation of
29118 *system()*. Although support for *system()* is not explicitly one of the goals for *posix_spawn()* and
29119 *posix_spawnp()*, it is covered under the “at least 50%” coverage goal.

29120 The intention is that the normal file descriptor inheritance across *fork()*, the subsequent effect of
29121 the specified spawn file actions, and the normal file descriptor inheritance across one of the *exec*
29122 family of functions should fully specify open file inheritance. The implementation need make no
29123 decisions regarding the set of open file descriptors when the child process image begins
29124 execution, those decisions having already been made by the caller and expressed as the set of
29125 open file descriptors and their FD_CLOEXEC flags at the time of the call and the spawn file
29126 actions object specified in the call. We have been assured that in cases where the POSIX
29127 *Start_Process* Ada primitives have been implemented in a library, this method of controlling file
29128 descriptor inheritance may be implemented very easily.

29129 We can identify several problems with *posix_spawn()* and *posix_spawnp()*, but there does not
29130 appear to be a solution that introduces fewer problems. Environment modification for child
29131 process attributes not specifiable via the *attrp* or *file_actions* arguments must be done in the
29132 parent process, and since the parent generally wants to save its context, it is more costly than
29133 similar functionality with *fork()/.exec*. It is also complicated to modify the environment of a
29134 multi-threaded process temporarily, since all threads must agree when it is safe for the
29135 environment to be changed. However, this cost is only borne by those invocations of
29136 *posix_spawn()* and *posix_spawnp()* that use the additional functionality. Since extensive
29137 modifications are not the usual case, and are particularly unlikely in time-critical code, keeping
29138 much of the environment control out of *posix_spawn()* and *posix_spawnp()* is appropriate design.

29139 The *posix_spawn()* and *posix_spawnp()* functions do not have all the power of *fork()/.exec*. This is
29140 to be expected. The *fork()* function is a wonderfully powerful operation. We do not expect to
29141 duplicate its functionality in a simple, fast function with no special hardware requirements. It is
29142 worth noting that *posix_spawn()* and *posix_spawnp()* are very similar to the process creation
29143 operations on many operating systems that are not UNIX systems.

29144 Requirements

29145 The requirements for *posix_spawn()* and *posix_spawnp()* are:

- 29146 • They must be implementable without an MMU or unusual hardware.
29147 • They must be compatible with existing POSIX standards.

29148 Additional goals are:

- 29149 • They should be efficiently implementable.
29150 • They should be able to replace at least 50% of typical executions of *fork()*.

- 29151 • A system with *posix_spawn()* and *posix_spawnp()* and without *fork()* should be useful, at least
29152 for realtime applications.
- 29153 • A system with *fork()* and the *exec* family should be able to implement *posix_spawn()* and
29154 *posix_spawnp()* as library routines.

29155 **Two-Syntax**

29156 POSIX *exec* has several calling sequences with approximately the same functionality. These
29157 appear to be required for compatibility with existing practice. Since the existing practice for the
29158 *posix_spawn**() functions is otherwise substantially unlike POSIX, we feel that simplicity
29159 outweighs compatibility. There are, therefore, only two names for the *posix_spawn**() functions.

29160 The parameter list does not differ between *posix_spawn()* and *posix_spawnp()*; *posix_spawnp()*
29161 interprets the second parameter more elaborately than *posix_spawn()*.

29162 **Compatibility with POSIX.5 (Ada)**

29163 The *Start_Process* and *Start_Process_Search* procedures from the *POSIX_Process_Primitives*
29164 package from the Ada language binding to POSIX.1 encapsulate *fork()* and *exec* functionality in a
29165 manner similar to that of *posix_spawn()* and *posix_spawnp()*. Originally, in keeping with our
29166 simplicity goal, the standard developers had limited the capabilities of *posix_spawn()* and
29167 *posix_spawnp()* to a subset of the capabilities of *Start_Process* and *Start_Process_Search*; certain
29168 non-default capabilities were not supported. However, based on suggestions by the ballot group
29169 to improve file descriptor mapping or drop it, and on the advice of an Ada Language Bindings
29170 working group member, the standard developers decided that *posix_spawn()* and *posix_spawnp()*
29171 should be sufficiently powerful to implement *Start_Process* and *Start_Process_Search*. The
29172 rationale is that if the Ada language binding to such a primitive had already been approved as
29173 an IEEE standard, there can be little justification for not approving the functionally-equivalent
29174 parts of a C binding. The only three capabilities provided by *posix_spawn()* and *posix_spawnp()*
29175 that are not provided by *Start_Process* and *Start_Process_Search* are optionally specifying the
29176 child's process group ID, the set of signals to be reset to default signal handling in the child
29177 process, and the child's scheduling policy and parameters.

29178 For the Ada language binding for *Start_Process* to be implemented with *posix_spawn()*, that
29179 binding would need to explicitly pass an empty signal mask and the parent's environment to
29180 *posix_spawn()* whenever the caller of *Start_Process* allowed these arguments to default, since
29181 *posix_spawn()* does not provide such defaults. The ability of *Start_Process* to mask user-specified
29182 signals during its execution is functionally unique to the Ada language binding and must be
29183 dealt with in the binding separately from the call to *posix_spawn()*.

29184 **Process Group**

29185 The process group inheritance field can be used to join the child process with an existing process
29186 group. By assigning a value of zero to the *spawn-pgroup* attribute of the object referenced by
29187 *attrp*, the *setpgid()* mechanism will place the child process in a new process group.

29188

Threads

29189

Without the *posix_spawn()* and *posix_spawnp()* 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_spawnp()* 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.

29190

29191

29192

29193

29194

29195

29196

29197

The *posix_spawn()* and *posix_spawnp()* 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.

29198

29199

29200

29201

29202

29203

Asynchronous Error Notification

29204

29205

29206

29207

29208

29209

A library implementation of *posix_spawn()* or *posix_spawnp()* 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()*.

29210

29211

29212

29213

29214

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_spawnp()* function has successfully returned.

29215

29216

29217

29218

29219

29220

29221

29222

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_spawnp()* 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_spawnp()* dependent on kernel modifications to *waitpid()* to be able to embed special information in *stat_val* to indicate a spawn failure.

29223

29224

29225

29226

29227

29228

29229

29230

29231

29232

29233

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_spawnp()* is permitted (and encouraged) to return any possible error as the function value, thus providing more detailed failure information to the parent process.

29234

29235

Thus, no special macros are available to isolate asynchronous *posix_spawn()* or *posix_spawnp()* errors. Instead, errors detected by the *posix_spawn()* or *posix_spawnp()* operations in the context

29236 of the child process before the new process image executes are reported by setting the child's
29237 exit status to 127. The calling process may use the WIFEXITED and WEXITSTATUS macros on
29238 the *stat_val* stored by the *wait()* or *waitpid()* functions to detect spawn failures to the extent that
29239 other status values with which the child process image may exit (before the parent can
29240 conclusively determine that the child process image has begun execution) are distinct from exit
29241 status 127.

29242 **FUTURE DIRECTIONS**

29243 None.

29244 **SEE ALSO**

29245 *alarm()*, *chmod()*, *close()*, *dup()*, *exec*, *exit()*, *fcntl()*, *fork()*, *kill()*, *open()*,
29246 *posix_spawn_file_actions_addclose()*, *posix_spawn_file_actions_adddup2()*,
29247 *posix_spawn_file_actions_addopen()*, *posix_spawn_file_actions_destroy()*, *posix_spawnattr_destroy()*,
29248 *posix_spawnattr_init()*, *posix_spawnattr_getsigdefault()*, *posix_spawnattr_getflags()*,
29249 *posix_spawnattr_getpgroup()*, *posix_spawnattr_getschedparam()*, *posix_spawnattr_getschedpolicy()*,
29250 *posix_spawnattr_getsigmask()*, *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setflags()*,
29251 *posix_spawnattr_setpgroup()*, *posix_spawnattr_setschedparam()*, *posix_spawnattr_setschedpolicy()*,
29252 *posix_spawnattr_setsigmask()*, *sched_setparam()*, *sched_setscheduler()*, *setpgid()*, *setuid()*, *stat()*,
29253 *times()*, *wait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**spawn.h**>

29254 **CHANGE HISTORY**

29255 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29256 IEEE PASC Interpretation 1003.1 #103 is applied, noting that the signal default actions are
29257 changed as well as the signal mask in step 2.

29258 IEEE PASC Interpretation 1003.1 #132 is applied.

29259 NAME

29260 posix_spawn_file_actions_addclose, posix_spawn_file_actions_addopen — add close or open
 29261 action to spawn file actions object (**ADVANCED REALTIME**)

29262 SYNOPSIS

29263 SPN #include <spawn.h>
 29264 int posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *
 29265 file_actions, int fildes);
 29266 int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *
 29267 restrict file_actions, int fildes,
 29268 const char *restrict path, int oflag, mode_t mode);
 29269

29270 DESCRIPTION

29271 These functions shall add or delete a close or open action to a spawn file actions object.

29272 A spawn file actions object is of type **posix_spawn_file_actions_t** (defined in <spawn.h>) and is
 29273 used to specify a series of actions to be performed by a *posix_spawn()* or *posix_spawnp()*
 29274 operation in order to arrive at the set of open file descriptors for the child process given the set of
 29275 open file descriptors of the parent. IEEE Std 1003.1-2001 does not define comparison or
 29276 assignment operators for the type **posix_spawn_file_actions_t**.

29277 A spawn file actions object, when passed to *posix_spawn()* or *posix_spawnp()*, shall specify how
 29278 the set of open file descriptors in the calling process is transformed into a set of potentially open
 29279 file descriptors for the spawned process. This transformation shall be as if the specified sequence
 29280 of actions was performed exactly once, in the context of the spawned process (prior to execution
 29281 of the new process image), in the order in which the actions were added to the object;
 29282 additionally, when the new process image is executed, any file descriptor (from this new set)
 29283 which has its FD_CLOEXEC flag set shall be closed (see *posix_spawn()*).

29284 The *posix_spawn_file_actions_addclose()* function shall add a *close* action to the object referenced
 29285 by *file_actions* that shall cause the file descriptor *fildes* to be closed (as if *close(fildes)* had been
 29286 called) when a new process is spawned using this file actions object.

29287 The *posix_spawn_file_actions_addopen()* function shall add an *open* action to the object referenced
 29288 by *file_actions* that shall cause the file named by *path* to be opened (as if *open(path, oflag, mode)*
 29289 had been called, and the returned file descriptor, if not *fildes*, had been changed to *fildes*) when a
 29290 new process is spawned using this file actions object. If *fildes* was already an open file descriptor,
 29291 it shall be closed before the new file is opened.

29292 The string described by *path* shall be copied by the *posix_spawn_file_actions_addopen()* function.

29293 RETURN VALUE

29294 Upon successful completion, these functions shall return zero; otherwise, an error number shall
 29295 be returned to indicate the error.

29296 ERRORS

29297 These functions shall fail if:

29298 [EBADF] The value specified by *fildes* is negative or greater than or equal to
 29299 {OPEN_MAX}.

29300 These functions may fail if:

29301 [EINVAL] The value specified by *file_actions* is invalid.

29302 [ENOMEM] Insufficient memory exists to add to the spawn file actions object.

29303 It shall not be considered an error for the *fdlist* argument passed to these functions to specify a
29304 file descriptor for which the specified operation could not be performed at the time of the call.
29305 Any such error will be detected when the associated file actions object is later used during a
29306 *posix_spawn()* or *posix_spawnp()* operation.

29307 EXAMPLES

29308 None.

29309 APPLICATION USAGE

29310 These functions are part of the Spawn option and need not be provided on all implementations.

29311 RATIONALE

29312 A spawn file actions object may be initialized to contain an ordered sequence of *close()*, *dup2()*,
29313 and *open()* operations to be used by *posix_spawn()* or *posix_spawnp()* to arrive at the set of open
29314 file descriptors inherited by the spawned process from the set of open file descriptors in the
29315 parent at the time of the *posix_spawn()* or *posix_spawnp()* call. It had been suggested that the
29316 *close()* and *dup2()* operations alone are sufficient to rearrange file descriptors, and that files
29317 which need to be opened for use by the spawned process can be handled either by having the
29318 calling process open them before the *posix_spawn()* or *posix_spawnp()* call (and close them after),
29319 or by passing filenames to the spawned process (in *argv*) so that it may open them itself. The
29320 standard developers recommend that applications use one of these two methods when practical,
29321 since detailed error status on a failed open operation is always available to the application this
29322 way. However, the standard developers feel that allowing a spawn file actions object to specify
29323 open operations is still appropriate because:

- 29324 1. It is consistent with equivalent POSIX.5 (Ada) functionality.
- 29325 2. It supports the I/O redirection paradigm commonly employed by POSIX programs
29326 designed to be invoked from a shell. When such a program is the child process, it may not
29327 be designed to open files on its own.
- 29328 3. It allows file opens that might otherwise fail or violate file ownership/access rights if
29329 executed by the parent process.

29330 Regarding 2. above, note that the spawn open file action provides to *posix_spawn()* and
29331 *posix_spawnp()* the same capability that the shell redirection operators provide to *system()*, only
29332 without the intervening execution of a shell; for example:

```
29333 system ("myprog <file1 3<file2");
```

29334 Regarding 3. above, note that if the calling process needs to open one or more files for access by
29335 the spawned process, but has insufficient spare file descriptors, then the open action is necessary
29336 to allow the *open()* to occur in the context of the child process after other file descriptors have
29337 been closed (that must remain open in the parent).

29338 Additionally, if a parent is executed from a file having a “set-user-id” mode bit set and the
29339 POSIX_SPAWN_RESETIDS flag is set in the spawn attributes, a file created within the parent
29340 process will (possibly incorrectly) have the parent’s effective user ID as its owner, whereas a file
29341 created via an *open()* action during *posix_spawn()* or *posix_spawnp()* will have the parent’s real
29342 ID as its owner; and an open by the parent process may successfully open a file to which the real
29343 user should not have access or fail to open a file to which the real user should have access.

29344

File Descriptor Mapping

29345

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_spawnp()* 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.

29346

29347

29348

29349

29350

29351

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.

29359 **FUTURE DIRECTIONS**

29360

None.

29361 **SEE ALSO**

29362

close(), *dup()*, *open()*, *posix_spawn()*, *posix_spawn_file_actions_adddup2()*,
posix_spawn_file_actions_destroy(), *posix_spawnp()*, the Base Definitions volume of
IEEE Std 1003.1-2001, <*spawn.h*>

29365 **CHANGE HISTORY**

29366

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29367

29368

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.

29369 NAME

29370 posix_spawn_file_actions_adddup2 — add dup2 action to spawn file actions object
 29371 (**ADVANCED REALTIME**)

29372 SYNOPSIS

29373 SPN #include <spawn.h>
 29374 int posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t *
 29375 file_actions, int fildes, int newfildes);
 29376

29377 DESCRIPTION

29378 The *posix_spawn_file_actions_adddup2()* function shall add a *dup2()* action to the object
 29379 referenced by *file_actions* that shall cause the file descriptor *fildes* to be duplicated as *newfildes* (as
 29380 if *dup2(fildes, newfildes)* had been called) when a new process is spawned using this file actions
 29381 object.

29382 A spawn file actions object is as defined in *posix_spawn_file_actions_addclose()*.

29383 RETURN VALUE

29384 Upon successful completion, the *posix_spawn_file_actions_adddup2()* function shall return zero;
 29385 otherwise, an error number shall be returned to indicate the error.

29386 ERRORS

29387 The *posix_spawn_file_actions_adddup2()* function shall fail if:
 29388 [EBADF] The value specified by *fildes* or *newfildes* is negative or greater than or equal to
 29389 {OPEN_MAX}.
 29390 [ENOMEM] Insufficient memory exists to add to the spawn file actions object.
 29391 The *posix_spawn_file_actions_adddup2()* function may fail if:
 29392 [EINVAL] The value specified by *file_actions* is invalid.
 29393 It shall not be considered an error for the *fildes* argument passed to the
 29394 *posix_spawn_file_actions_adddup2()* function to specify a file descriptor for which the specified
 29395 operation could not be performed at the time of the call. Any such error will be detected when
 29396 the associated file actions object is later used during a *posix_spawn()* or *posix_spawnp()*
 29397 operation.

29398 EXAMPLES

29399 None.

29400 APPLICATION USAGE

29401 The *posix_spawn_file_actions_adddup2()* function is part of the Spawn option and need not be
 29402 provided on all implementations.

29403 RATIONALE

29404 Refer to the RATIONALE in *posix_spawn_file_actions_addclose()*.

29405 FUTURE DIRECTIONS

29406 None.

29407 SEE ALSO

29408 *dup()*, *posix_spawn()*, *posix_spawn_file_actions_addclose()*, *posix_spawn_file_actions_destroy()*,
 29409 *posix_spawnp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <spawn.h>

29410 CHANGE HISTORY

29411 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29412 IEEE PASC Interpretation 1003.1 #104 is applied, noting that the [EBADF] error can apply to the *newfdes* argument in addition to *fdes*.

29414 **NAME**

29415 **posix_spawn_file_actions_addopen** — add open action to spawn file actions object
29416 **(ADVANCED REALTIME)**

29417 **SYNOPSIS**

29418 SPN

```
#include <spawn.h>
```

29419

```
int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *
29420                                          restrict file_actions, int fildes,
29421                                          const char *restrict path, int oflag, mode_t mode);
```

29422

29423 **DESCRIPTION**

29424 Refer to *posix_spawn_file_actions_addclose()*.

29425 NAME

29426 **posix_spawn_file_actions_destroy**, **posix_spawn_file_actions_init** — destroy and initialize
29427 spawn file actions object (**ADVANCED REALTIME**)

29428 SYNOPSIS

29429 SPN

```
#include <spawn.h>
```


29430 int posix_spawn_file_actions_destroy(posix_spawn_file_actions_t *
29431 file_actions);
29432 int posix_spawn_file_actions_init(posix_spawn_file_actions_t *
29433 file_actions);
29434

29435 DESCRIPTION

29436 The *posix_spawn_file_actions_destroy()* function shall destroy the object referenced by *file_actions*;
29437 the object becomes, in effect, uninitialized. An implementation may cause
29438 *posix_spawn_file_actions_destroy()* to set the object referenced by *file_actions* to an invalid value. A
29439 destroyed spawn file actions object can be reinitialized using *posix_spawn_file_actions_init()*; the
29440 results of otherwise referencing the object after it has been destroyed are undefined.

29441 The *posix_spawn_file_actions_init()* function shall initialize the object referenced by *file_actions* to
29442 contain no file actions for *posix_spawn()* or *posix_spawnp()* to perform.

29443 A spawn file actions object is as defined in *posix_spawn_file_actions_addclose()*.

29444 The effect of initializing an already initialized spawn file actions object is undefined.

29445 RETURN VALUE

29446 Upon successful completion, these functions shall return zero; otherwise, an error number shall
29447 be returned to indicate the error.

29448 ERRORS

29449 The *posix_spawn_file_actions_init()* function shall fail if:

29450 [ENOMEM] Insufficient memory exists to initialize the spawn file actions object.

29451 The *posix_spawn_file_actions_destroy()* function may fail if:

29452 [EINVAL] The value specified by *file_actions* is invalid.

29453 EXAMPLES

29454 None.

29455 APPLICATION USAGE

29456 These functions are part of the Spawn option and need not be provided on all implementations.

29457 RATIONALE

29458 Refer to the RATIONALE in *posix_spawn_file_actions_addclose()*.

29459 FUTURE DIRECTIONS

29460 None.

29461 SEE ALSO

29462 *posix_spawn()*, *posix_spawnp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**spawn.h**>

29463 CHANGE HISTORY

29464 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29465 In the SYNOPSIS, the inclusion of <**sys/types.h**> is no longer required.

29466 NAME

29467 **posix_spawnattr_destroy**, **posix_spawnattr_init** — destroy and initialize spawn attributes object
29468 **(ADVANCED REALTIME)**

29469 SYNOPSIS

29470 SPN

```
#include <spawn.h>
```


29471 int **posix_spawnattr_destroy**(**posix_spawnattr_t** *attr);
29472 int **posix_spawnattr_init**(**posix_spawnattr_t** *attr);
29473

29474 DESCRIPTION

29475 The **posix_spawnattr_destroy()** function shall destroy a spawn attributes object. A destroyed *attr*
29476 attributes object can be reinitialized using **posix_spawnattr_init()**; the results of otherwise
29477 referencing the object after it has been destroyed are undefined. An implementation may cause
29478 **posix_spawnattr_destroy()** to set the object referenced by *attr* to an invalid value.

29479 The **posix_spawnattr_init()** function shall initialize a spawn attributes object *attr* with the default
29480 value for all of the individual attributes used by the implementation. Results are undefined if
29481 **posix_spawnattr_init()** is called specifying an already initialized *attr* attributes object.

29482 A spawn attributes object is of type **posix_spawnattr_t** (defined in <spawn.h>) and is used to
29483 specify the inheritance of process attributes across a spawn operation. IEEE Std 1003.1-2001 does
29484 not define comparison or assignment operators for the type **posix_spawnattr_t**.

29485 Each implementation shall document the individual attributes it uses and their default values
29486 unless these values are defined by IEEE Std 1003.1-2001. Attributes not defined by
29487 IEEE Std 1003.1-2001, their default values, and the names of the associated functions to get and
29488 set those attribute values are implementation-defined.

29489 The resulting spawn attributes object (possibly modified by setting individual attribute values),
29490 is used to modify the behavior of **posix_spawn()** or **posix_spawnp()**. After a spawn attributes
29491 object has been used to spawn a process by a call to a **posix_spawn()** or **posix_spawnp()**, any
29492 function affecting the attributes object (including destruction) shall not affect any process that
29493 has been spawned in this way.

29494 RETURN VALUE

29495 Upon successful completion, **posix_spawnattr_destroy()** and **posix_spawnattr_init()** shall return
29496 zero; otherwise, an error number shall be returned to indicate the error.

29497 ERRORS

29498 The **posix_spawnattr_init()** function shall fail if:
29499 [ENOMEM] Insufficient memory exists to initialize the spawn attributes object.
29500 The **posix_spawnattr_destroy()** function may fail if:
29501 [EINVAL] The value specified by attr is invalid.

29502 EXAMPLES

29503 None.

29504 APPLICATION USAGE

29505 These functions are part of the Spawn option and need not be provided on all implementations.

29506 RATIONALE

29507 The original spawn interface proposed in IEEE Std 1003.1-2001 defined the attributes that specify
29508 the inheritance of process attributes across a spawn operation as a structure. In order to be able
29509 to separate optional individual attributes under their appropriate options (that is, the *spawn-*
29510 *schedparam* and *spawn-schedpolicy* attributes depending upon the Process Scheduling option), and

29511 also for extensibility and consistency with the newer POSIX interfaces, the attributes interface
29512 has been changed to an opaque data type. This interface now consists of the type
29513 **posix_spawnattr_t**, representing a spawn attributes object, together with associated functions to
29514 initialize or destroy the attributes object, and to set or get each individual attribute. Although the
29515 new object-oriented interface is more verbose than the original structure, it is simple to use,
29516 more extensible, and easy to implement.

29517 FUTURE DIRECTIONS

29518 None.

29519 SEE ALSO

29520 *posix_spawn()*, *posix_spawnattr_getsigdefault()*, *posix_spawnattr_getflags()*,
29521 *posix_spawnattr_getpgroup()*, *posix_spawnattr_getschedparam()*, *posix_spawnattr_getschedpolicy()*,
29522 *posix_spawnattr_getsigmask()*, *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setflags()*,
29523 *posix_spawnattr_setpgroup()*, *posix_spawnattr_setsigmask()*, *posix_spawnattr_setschedpolicy()*,
29524 *posix_spawnattr_setschedparam()*, *posix_spawnp()*, the Base Definitions volume of
29525 IEEE Std 1003.1-2001, <**spawn.h**>

29526 CHANGE HISTORY

29527 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29528 IEEE PASC Interpretation 1003.1 #106 is applied, noting that the effect of initializing an already
29529 initialized spawn attributes option is undefined.

29530 NAME

29531 posix_spawnattr_getflags, posix_spawnattr_setflags — get and set the spawn-flags attribute of a
29532 spawn attributes object (**ADVANCED REALTIME**)

29533 SYNOPSIS

29534 SPN #include <spawn.h>

29535 int posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
29536 short *restrict flags);
29537 int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
29538

29539 DESCRIPTION

29540 The *posix_spawnattr_getflags()* function shall obtain the value of the *spawn-flags* attribute from
29541 the attributes object referenced by *attr*.

29542 The *posix_spawnattr_setflags()* function shall set the *spawn-flags* attribute in an initialized
29543 attributes object referenced by *attr*.

29544 The *spawn-flags* attribute is used to indicate which process attributes are to be changed in the
29545 new process image when invoking *posix_spawn()* or *posix_spawnp()*. It is the bitwise-inclusive
29546 OR of zero or more of the following flags:

29547 POSIX_SPAWN_RESETIDS
29548 POSIX_SPAWN_SETPGROUP
29549 POSIX_SPAWN_SETSIGDEF
29550 POSIX_SPAWN_SETSIGMASK
29551 PS POSIX_SPAWN_SETSCHEDPARAM
29552 POSIX_SPAWN_SETSCHEDULER
29553

29554 These flags are defined in <spawn.h>. The default value of this attribute shall be as if no flags
29555 were set.

29556 RETURN VALUE

29557 Upon successful completion, *posix_spawnattr_getflags()* shall return zero and store the value of
29558 the *spawn-flags* attribute of *attr* into the object referenced by the *flags* parameter; otherwise, an
29559 error number shall be returned to indicate the error.

29560 Upon successful completion, *posix_spawnattr_setflags()* shall return zero; otherwise, an error
29561 number shall be returned to indicate the error.

29562 ERRORS

29563 These functions may fail if:

29564 [EINVAL] The value specified by *attr* is invalid.

29565 The *posix_spawnattr_setflags()* function may fail if:

29566 [EINVAL] The value of the attribute being set is not valid.

29567 EXAMPLES

29568 None.

29569 APPLICATION USAGE

29570 These functions are part of the Spawn option and need not be provided on all implementations.

29571 RATIONALE

29572 None.

29573 FUTURE DIRECTIONS

29574 None.

29575 SEE ALSO

29576 *posix_spawn()*, *posix_spawnattr_destroy()*, *posix_spawnattr_init()*, *posix_spawnattr_getsigdefault()*,
29577 *posix_spawnattr_getpgroup()*, *posix_spawnattr_getschedparam()*, *posix_spawnattr_getschedpolicy()*,
29578 *posix_spawnattr_getsigmask()*, *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setpgroup()*,
29579 *posix_spawnattr_setschedparam()*, *posix_spawnattr_setschedpolicy()*, *posix_spawnattr_setsigmask()*,
29580 *posix_spawnp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**spawn.h**>

29581 CHANGE HISTORY

29582 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29583 NAME

29584 posix_spawnattr_getpgroup, posix_spawnattr_setpgroup — get and set the spawn-pgroup
 29585 attribute of a spawn attributes object (**ADVANCED REALTIME**)

29586 SYNOPSIS

```
29587 SPN #include <spawn.h>
29588
29589     int posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr,
29590             pid_t *restrict pgroup);
29591     int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

29592 DESCRIPTION

29593 The *posix_spawnattr_getpgroup()* function shall obtain the value of the *spawn-pgroup* attribute
 29594 from the attributes object referenced by *attr*.

29595 The *posix_spawnattr_setpgroup()* function shall set the *spawn-pgroup* attribute in an initialized
 29596 attributes object referenced by *attr*.

29597 The *spawn-pgroup* attribute represents the process group to be joined by the new process image
 29598 in a spawn operation (if *POSIX_SPAWN_SETPGROUP* is set in the *spawn-flags* attribute). The
 29599 default value of this attribute shall be zero.

29600 RETURN VALUE

29601 Upon successful completion, *posix_spawnattr_getpgroup()* shall return zero and store the value of
 29602 the *spawn-pgroup* attribute of *attr* into the object referenced by the *pgroup* parameter; otherwise,
 29603 an error number shall be returned to indicate the error.

29604 Upon successful completion, *posix_spawnattr_setpgroup()* shall return zero; otherwise, an error
 29605 number shall be returned to indicate the error.

29606 ERRORS

29607 These functions may fail if:

29608 [EINVAL] The value specified by *attr* is invalid.

29609 The *posix_spawnattr_setpgroup()* function may fail if:

29610 [EINVAL] The value of the attribute being set is not valid.

29611 EXAMPLES

29612 None.

29613 APPLICATION USAGE

29614 These functions are part of the Spawn option and need not be provided on all implementations.

29615 RATIONALE

29616 None.

29617 FUTURE DIRECTIONS

29618 None.

29619 SEE ALSO

29620 *posix_spawn()*, *posix_spawnattr_destroy()*, *posix_spawnattr_init()*, *posix_spawnattr_getsigdefault()*,
 29621 *posix_spawnattr_getflags()*, *posix_spawnattr_getschedparam()*, *posix_spawnattr_getschedpolicy()*,
 29622 *posix_spawnattr_getsigmask()*, *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setflags()*,
 29623 *posix_spawnattr_setschedparam()*, *posix_spawnattr_setschedpolicy()*, *posix_spawnattr_setsigmask()*,
 29624 *posix_spawnp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <spawn.h>

29625 CHANGE HISTORY

29626 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29627 NAME

29628 **posix_spawnattr_getschedparam**, **posix_spawnattr_setschedparam** — get and set the spawn-
 29629 schedparam attribute of a spawn attributes object (**ADVANCED REALTIME**)

29630 SYNOPSIS

```
29631 SPN PS #include <spawn.h>
29632      #include <sched.h>
29633
29634      int posix_spawnattr_getschedparam(const posix_spawnattr_t *
29635          restrict attr, struct sched_param *restrict schedparam);
29636      int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
29637          const struct sched_param *restrict schedparam);
29638
```

29638 DESCRIPTION

29639 The *posix_spawnattr_getschedparam()* function shall obtain the value of the *spawn-schedparam*
 29640 attribute from the attributes object referenced by *attr*.

29641 The *posix_spawnattr_setschedparam()* function shall set the *spawn-schedparam* attribute in an
 29642 initialized attributes object referenced by *attr*.

29643 The *spawn-schedparam* attribute represents the scheduling parameters to be assigned to the new
 29644 process image in a spawn operation (if **POSIX_SPAWN_SETSCHEDULER** or
 29645 **POSIX_SPAWN_SETSCHEDPARAM** is set in the *spawn-flags* attribute). The default value of this
 29646 attribute is unspecified.

29647 RETURN VALUE

29648 Upon successful completion, *posix_spawnattr_getschedparam()* shall return zero and store the
 29649 value of the *spawn-schedparam* attribute of *attr* into the object referenced by the *schedparam*
 29650 parameter; otherwise, an error number shall be returned to indicate the error.

29651 Upon successful completion, *posix_spawnattr_setschedparam()* shall return zero; otherwise, an
 29652 error number shall be returned to indicate the error.

29653 ERRORS

29654 These functions may fail if:

29655 [EINVAL] The value specified by *attr* is invalid.

29656 The *posix_spawnattr_setschedparam()* function may fail if:

29657 [EINVAL] The value of the attribute being set is not valid.

29658 EXAMPLES

29659 None.

29660 APPLICATION USAGE

29661 These functions are part of the Spawn and Process Scheduling options and need not be provided
 29662 on all implementations.

29663 RATIONALE

29664 None.

29665 FUTURE DIRECTIONS

29666 None.

29667 SEE ALSO

29668 *posix_spawn()*, *posix_spawnattr_destroy()*, *posix_spawnattr_init()*, *posix_spawnattr_getsigdefault()*,
 29669 *posix_spawnattr_getflags()*, *posix_spawnattr_getpgroup()*, *posix_spawnattr_getschedpolicy()*,
 29670 *posix_spawnattr_getsigmask()*, *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setflags()*,

29671 *posix_spawnattr_setpgroup(), posix_spawnattr_setschedpolicy(), posix_spawnattr_setsigmask(),*
29672 *posix_spawnp(), the Base Definitions volume of IEEE Std 1003.1-2001, <sched.h>, <spawn.h>*

29673 CHANGE HISTORY

29674 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29675 NAME

29676 **posix_spawnattr_getschedpolicy**, **posix_spawnattr_setschedpolicy** — get and set the spawn-
 29677 schedpolicy attribute of a spawn attributes object (**ADVANCED REALTIME**)

29678 SYNOPSIS

```
29679 SPN PS #include <spawn.h>
29680      #include <sched.h>
29681      int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *
29682          restrict attr, int *restrict schedpolicy);
29683      int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
29684          int schedpolicy);
29685
```

29686 DESCRIPTION

29687 The *posix_spawnattr_getschedpolicy()* function shall obtain the value of the *spawn-schedpolicy*
 29688 attribute from the attributes object referenced by *attr*.

29689 The *posix_spawnattr_setschedpolicy()* function shall set the *spawn-schedpolicy* attribute in an
 29690 initialized attributes object referenced by *attr*.

29691 The *spawn-schedpolicy* attribute represents the scheduling policy to be assigned to the new
 29692 process image in a spawn operation (if **POSIX_SPAWN_SETSCHEDULER** is set in the *spawn-*
 29693 *flags* attribute). The default value of this attribute is unspecified.

29694 RETURN VALUE

29695 Upon successful completion, *posix_spawnattr_getschedpolicy()* shall return zero and store the
 29696 value of the *spawn-schedpolicy* attribute of *attr* into the object referenced by the *schedpolicy*
 29697 parameter; otherwise, an error number shall be returned to indicate the error.

29698 Upon successful completion, *posix_spawnattr_setschedpolicy()* shall return zero; otherwise, an
 29699 error number shall be returned to indicate the error.

29700 ERRORS

29701 These functions may fail if:

29702 [EINVAL] The value specified by *attr* is invalid.

29703 The *posix_spawnattr_setschedpolicy()* function may fail if:

29704 [EINVAL] The value of the attribute being set is not valid.

29705 EXAMPLES

29706 None.

29707 APPLICATION USAGE

29708 These functions are part of the Spawn and Process Scheduling options and need not be provided
 29709 on all implementations.

29710 RATIONALE

29711 None.

29712 FUTURE DIRECTIONS

29713 None.

29714 SEE ALSO

29715 **posix_spawn()**, **posix_spawnattr_destroy()**, **posix_spawnattr_init()**, **posix_spawnattr_getsigdefault()**,
 29716 **posix_spawnattr_getflags()**, **posix_spawnattr_getpgroup()**, **posix_spawnattr_getschedparam()**,
 29717 **posix_spawnattr_getsigmask()**, **posix_spawnattr_setsigdefault()**, **posix_spawnattr_setflags()**,
 29718 **posix_spawnattr_setpgroup()**, **posix_spawnattr_setschedparam()**, **posix_spawnattr_setsigmask()**,

29719 *posix_spawnp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sched.h>, <spawn.h>

29720 CHANGE HISTORY

29721 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29722 NAME

29723 **posix_spawnattr_getsigdefault**, **posix_spawnattr_setsigdefault** — get and set the spawn-
 29724 sigdefault attribute of a spawn attributes object (**ADVANCED REALTIME**)

29725 SYNOPSIS

```
29726 SPN #include <signal.h>
29727      #include <spawn.h>
29728
29729      int posix_spawnattr_getsigdefault(const posix_spawnattr_t *
29730          restrict attr, sigset_t *restrict sigdefault);
29731      int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
29732          const sigset_t *restrict sigdefault);
```

29733 DESCRIPTION

29734 The *posix_spawnattr_getsigdefault()* function shall obtain the value of the *spawn-sigdefault*
 29735 attribute from the attributes object referenced by *attr*.

29736 The *posix_spawnattr_setsigdefault()* function shall set the *spawn-sigdefault* attribute in an
 29737 initialized attributes object referenced by *attr*.

29738 The *spawn-sigdefault* attribute represents the set of signals to be forced to default signal handling
 29739 in the new process image (if **POSIX_SPAWN_SETSIGDEF** is set in the *spawn-flags* attribute) by a
 29740 spawn operation. The default value of this attribute shall be an empty signal set.

29741 RETURN VALUE

29742 Upon successful completion, *posix_spawnattr_getsigdefault()* shall return zero and store the value
 29743 of the *spawn-sigdefault* attribute of *attr* into the object referenced by the *sigdefault* parameter;
 29744 otherwise, an error number shall be returned to indicate the error.

29745 Upon successful completion, *posix_spawnattr_setsigdefault()* shall return zero; otherwise, an error
 29746 number shall be returned to indicate the error.

29747 ERRORS

29748 These functions may fail if:

29749 [EINVAL] The value specified by *attr* is invalid.

29750 The *posix_spawnattr_setsigdefault()* function may fail if:

29751 [EINVAL] The value of the attribute being set is not valid.

29752 EXAMPLES

29753 None.

29754 APPLICATION USAGE

29755 These functions are part of the Spawn option and need not be provided on all implementations.

29756 RATIONALE

29757 None.

29758 FUTURE DIRECTIONS

29759 None.

29760 SEE ALSO

29761 **posix_spawn()**, **posix_spawnattr_destroy()**, **posix_spawnattr_init()**, **posix_spawnattr_getflags()**,
 29762 **posix_spawnattr_getpgroup()**, **posix_spawnattr_getschedparam()**, **posix_spawnattr_getschedpolicy()**,
 29763 **posix_spawnattr_getsigmask()**, **posix_spawnattr_setflags()**, **posix_spawnattr_setpgroup()**,
 29764 **posix_spawnattr_setschedparam()**, **posix_spawnattr_setschedpolicy()**, **posix_spawnattr_setsigmask()**,
 29765 **posix_spawnp()**, the Base Definitions volume of IEEE Std 1003.1-2001, **<signal.h>**, **<spawn.h>**

29766 CHANGE HISTORY

29767 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29768 NAME

29769 posix_spawnattr_getsigmask, posix_spawnattr_setsigmask — get and set the spawn-sigmask
 29770 attribute of a spawn attributes object (**ADVANCED REALTIME**)

29771 SYNOPSIS

```
29772 SPN #include <signal.h>
29773      #include <spawn.h>
29774
29775      int posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict attr,
29776                                         sigset_t *restrict sigmask);
29777      int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
29778                                         const sigset_t *restrict sigmask);
```

29779 DESCRIPTION

29780 The *posix_spawnattr_getsigmask()* function shall obtain the value of the *spawn-sigmask* attribute
 29781 from the attributes object referenced by *attr*.

29782 The *posix_spawnattr_setsigmask()* function shall set the *spawn-sigmask* attribute in an initialized
 29783 attributes object referenced by *attr*.

29784 The *spawn-sigmask* attribute represents the signal mask in effect in the new process image of a
 29785 spawn operation (if **POSIX_SPAWN_SETSIGMASK** is set in the *spawn-flags* attribute). The
 29786 default value of this attribute is unspecified.

29787 RETURN VALUE

29788 Upon successful completion, *posix_spawnattr_getsigmask()* shall return zero and store the value
 29789 of the *spawn-sigmask* attribute of *attr* into the object referenced by the *sigmask* parameter;
 29790 otherwise, an error number shall be returned to indicate the error.

29791 Upon successful completion, *posix_spawnattr_setsigmask()* shall return zero; otherwise, an error
 29792 number shall be returned to indicate the error.

29793 ERRORS

29794 These functions may fail if:

29795 [EINVAL] The value specified by *attr* is invalid.

29796 The *posix_spawnattr_setsigmask()* function may fail if:

29797 [EINVAL] The value of the attribute being set is not valid.

29798 EXAMPLES

29799 None.

29800 APPLICATION USAGE

29801 These functions are part of the Spawn option and need not be provided on all implementations.

29802 RATIONALE

29803 None.

29804 FUTURE DIRECTIONS

29805 None.

29806 SEE ALSO

29807 *posix_spawn()*, *posix_spawnattr_destroy()*, *posix_spawnattr_init()*, *posix_spawnattr_getsigdefault()*,
 29808 *posix_spawnattr_getflags()*, *posix_spawnattr_getpgroup()*, *posix_spawnattr_getschedparam()*,
 29809 *posix_spawnattr_getschedpolicy()*, *posix_spawnattr_setsigdefault()*, *posix_spawnattr_setflags()*,
 29810 *posix_spawnattr_setpgroup()*, *posix_spawnattr_setschedparam()*, *posix_spawnattr_setschedpolicy()*,
 29811 *posix_spawnp()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<signal.h>**, **<spawn.h>**

29812 CHANGE HISTORY

29813 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

29814 NAME

29815 **posix_spawnattr_init** — initialize the spawn attributes object (**ADVANCED REALTIME**)

29816 SYNOPSIS

29817 SPN `#include <spawn.h>`

29818 `int posix_spawnattr_init(posix_spawnattr_t *attr);`

29819

29820 DESCRIPTION

29821 Refer to *posix_spawnattr_destroy()*.

29822 **NAME**29823 **posix_spawnattr_setflags** — set the spawn-flags attribute of a spawn attributes object
29824 **(ADVANCED REALTIME)**29825 **SYNOPSIS**29826 SPN

```
#include <spawn.h>
```


29827

```
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```


2982829829 **DESCRIPTION**29830 Refer to *posix_spawnattr_getflags()*.

29831 NAME

29832 **posix_spawnattr_setpgroup** — set the spawn-pgroup attribute of a spawn attributes object
29833 **(ADVANCED REALTIME)**

29834 SYNOPSIS

29835 SPN `#include <spawn.h>`
29836 `int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);`
29837

29838 DESCRIPTION

29839 Refer to *posix_spawnattr_getpgroup()*.

29840 NAME

29841 **posix_spawnattr_setschedparam** — set the spawn-schedparam attribute of a spawn attributes
29842 object (**ADVANCED REALTIME**)

29843 SYNOPSIS

```
29844 SPN PS #include <sched.h>
29845      #include <spawn.h>
29846      int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
29847          const struct sched_param *restrict schedparam);
```

29849 DESCRIPTION

29850 Refer to *posix_spawnattr_getschedparam()*.

29851 NAME

29852 **posix_spawnattr_setschedpolicy** — set the spawn-schedpolicy attribute of a spawn attributes
29853 object (**ADVANCED REALTIME**)

29854 SYNOPSIS

```
29855 SPN PS #include <sched.h>
29856      #include <spawn.h>
29857      int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
29858          int schedpolicy);
```

29860 DESCRIPTION

29861 Refer to *posix_spawnattr_getschedpolicy()*.

29862 NAME

29863 posix_spawnattr_setsigdefault — set the spawn-sigdefault attribute of a spawn attributes object
29864 (**ADVANCED REALTIME**)

29865 SYNOPSIS

```
29866 SPN #include <signal.h>
29867     #include <spawn.h>
29868     int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
29869             const sigset_t *restrict sigdefault);
29870
```

29871 DESCRIPTION

29872 Refer to *posix_spawnattr_getsigdefault()*.

29873 NAME

29874 posix_spawnattr_setsigmask — set the spawn-sigmask attribute of a spawn attributes object
29875 (**ADVANCED REALTIME**)

29876 SYNOPSIS

```
29877 SPN #include <signal.h>
29878     #include <spawn.h>
29879     int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
29880             const sigset_t *restrict sigmask);
29881
```

29882 DESCRIPTION

29883 Refer to *posix_spawnattr_getsigmask()*.

29884 NAME

29885 **posix_spawnp** — spawn a process (**ADVANCED REALTIME**)

29886 SYNOPSIS

29887 SPN `#include <spawn.h>`

```
29888       int posix_spawnp(pid_t *restrict pid, const char *restrict file,
29889                   const posix_spawn_file_actions_t *file_actions,
29890                   const posix_spawnattr_t *restrict attrp,
29891                   char *const argv[restrict], char *const envp[restrict]);
```

29892

29893 DESCRIPTION

29894 Refer to *posix_spawn()*.

29895 **NAME**

29896 `posix_trace_attr_destroy`, `posix_trace_attr_init` — destroy and initialize the trace stream
29897 attributes object (**TRACING**)

29898 **SYNOPSIS**

29899 TRC `#include <trace.h>`
29900 `int posix_trace_attr_destroy(trace_attr_t *attr);`
29901 `int posix_trace_attr_init(trace_attr_t *attr);`
29902

29903 **DESCRIPTION**

29904 The `posix_trace_attr_destroy()` function shall destroy an initialized trace attributes object. A
29905 destroyed `attr` attributes object can be reinitialized using `posix_trace_attr_init()`; the results of
29906 otherwise referencing the object after it has been destroyed are undefined.

29907 The `posix_trace_attr_init()` function shall initialize a trace attributes object `attr` with the default
29908 value for all of the individual attributes used by a given implementation. The read-only
29909 *generation-version* and *clock-resolution* attributes of the newly initialized trace attributes object
29910 shall be set to their appropriate values (see Section 2.11.1.2 (on page 76)).

29911 Results are undefined if `posix_trace_attr_init()` is called specifying an already initialized `attr`
29912 attributes object.

29913 Implementations may add extensions to the trace attributes object structure as permitted in the
29914 Base Definitions volume of IEEE Std 1003.1-2001, Chapter 2, Conformance.

29915 The resulting attributes object (possibly modified by setting individual attributes values), when
29916 used by `posix_trace_create()`, defines the attributes of the trace stream created. A single attributes
29917 object can be used in multiple calls to `posix_trace_create()`. After one or more trace streams have
29918 been created using an attributes object, any function affecting that attributes object, including
29919 destruction, shall not affect any trace stream previously created. An initialized attributes object
29920 also serves to receive the attributes of an existing trace stream or trace log when calling the
29921 `posix_trace_get_attr()` function.

29922 **RETURN VALUE**

29923 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
29924 return the corresponding error number.

29925 **ERRORS**

29926 The `posix_trace_attr_destroy()` function may fail if:

29927 [EINVAL] The value of `attr` is invalid.

29928 The `posix_trace_attr_init()` function shall fail if:

29929 [ENOMEM] Insufficient memory exists to initialize the trace attributes object.

29930 **EXAMPLES**

29931 None.

29932 **APPLICATION USAGE**

29933 None.

29934 **RATIONALE**

29935 None.

29936 FUTURE DIRECTIONS

29937 None.

29938 SEE ALSO

29939 *posix_trace_create()*, *posix_trace_get_attr()*, *uname()*, the Base Definitions volume of
29940 IEEE Std 1003.1-2001, <trace.h>

29941 CHANGE HISTORY

29942 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

29943 IEEE PASC Interpretation 1003.1 #123 is applied.

29944 NAME

29945 posix_trace_attr_getclockres, posix_trace_attr_getcreatetime, posix_trace_attr_getgenversion,
 29946 posix_trace_attr_getname, posix_trace_attr_setname — retrieve and set information about a
 29947 trace stream (TRACING)

29948 SYNOPSIS

```
29949 TRC #include <time.h>
29950      #include <trace.h>

29951      int posix_trace_attr_getclockres(const trace_attr_t *attr,
29952          struct timespec *resolution);
29953      int posix_trace_attr_getcreatetime(const trace_attr_t *attr,
29954          struct timespec *createtime);

29955      #include <trace.h>

29956      int posix_trace_attr_getgenversion(const trace_attr_t *attr,
29957          char *genversion);
29958      int posix_trace_attr_getname(const trace_attr_t *attr,
29959          char *tracename);
29960      int posix_trace_attr_setname(trace_attr_t *attr,
29961          const char *tracename);

29962
```

29963 DESCRIPTION

29964 The *posix_trace_attr_getclockres()* function shall copy the clock resolution of the clock used to
 29965 generate timestamps from the *clock-resolution* attribute of the attributes object pointed to by the
 29966 *attr* argument into the structure pointed to by the *resolution* argument.

29967 The *posix_trace_attr_getcreatetime()* function shall copy the trace stream creation time from the
 29968 *creation-time* attribute of the attributes object pointed to by the *attr* argument into the structure
 29969 pointed to by the *createtime* argument. The *creation-time* attribute shall represent the time of
 29970 creation of the trace stream.

29971 The *posix_trace_attr_getgenversion()* function shall copy the string containing version information
 29972 from the *generation-version* attribute of the attributes object pointed to by the *attr* argument into
 29973 the string pointed to by the *genversion* argument. The *genversion* argument shall be the address of
 29974 a character array which can store at least {TRACE_NAME_MAX} characters.

29975 The *posix_trace_attr_getname()* function shall copy the string containing the trace name from the
 29976 *trace-name* attribute of the attributes object pointed to by the *attr* argument into the string
 29977 pointed to by the *tracename* argument. The *tracename* argument shall be the address of a character
 29978 array which can store at least {TRACE_NAME_MAX} characters.

29979 The *posix_trace_attr_setname()* function shall set the name in the *trace-name* attribute of the
 29980 attributes object pointed to by the *attr* argument, using the trace name string supplied by the
 29981 *tracename* argument. If the supplied string contains more than {TRACE_NAME_MAX}
 29982 characters, the name copied into the *trace-name* attribute may be truncated to one less than the
 29983 length of {TRACE_NAME_MAX} characters. The default value is a null string.

29984 RETURN VALUE

29985 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 29986 return the corresponding error number.

29987 If successful, the *posix_trace_attr_getclockres()* function stores the *clock-resolution* attribute value
 29988 in the object pointed to by *resolution*. Otherwise, the content of this object is unspecified.

29989 If successful, the *posix_trace_attr_getcreatetime()* function stores the trace stream creation time in
29990 the object pointed to by *createtime*. Otherwise, the content of this object is unspecified.

29991 If successful, the *posix_trace_attr_getgenversion()* function stores the trace version information in
29992 the string pointed to by *genversion*. Otherwise, the content of this string is unspecified.

29993 If successful, the *posix_trace_attr_getname()* function stores the trace name in the string pointed
29994 to by *tracename*. Otherwise, the content of this string is unspecified.

29995 **ERRORS**

29996 The *posix_trace_attr_getclockres()*, *posix_trace_attr_getcreatetime()*, *posix_trace_attr_getgenversion()*,
29997 and *posix_trace_attr_getname()* functions may fail if:

29998 [EINVAL] The value specified by one of the arguments is invalid.

29999 **EXAMPLES**

30000 None.

30001 **APPLICATION USAGE**

30002 None.

30003 **RATIONALE**

30004 None.

30005 **FUTURE DIRECTIONS**

30006 None.

30007 **SEE ALSO**

30008 *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_get_attr()*, *uname()*, the Base Definitions
30009 volume of IEEE Std 1003.1-2001, <time.h>, <trace.h>

30010 **CHANGE HISTORY**

30011 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30012 NAME

30013 posix_trace_attr_getinherited, posix_trace_attr_getlogfullpolicy,
 30014 posix_trace_attr_getstreamfullpolicy, posix_trace_attr_setinherited,
 30015 posix_trace_attr_setlogfullpolicy, posix_trace_attr_setstreamfullpolicy — retrieve and set the
 30016 behavior of a trace stream (**TRACING**)

30017 SYNOPSIS

```
30018 TRC #include <trace.h>
30019 TRC TRI int posix_trace_attr_getinherited(const trace_attr_t *restrict attr,
30020            int *restrict inheritancepolicy);
30021 TRL int posix_trace_attr_getlogfullpolicy(const trace_attr_t *restrict attr,
30022            int *restrict logpolicy);
30023 TRC int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *attr,
30024            int *stremppolicy);
30025 TRC TRI int posix_trace_attr_setinherited(trace_attr_t *attr,
30026            int inheritancepolicy);
30027 TRL int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
30028            int logpolicy);
30029 TRC int posix_trace_attr_setstreamfullpolicy(trace_attr_t *attr,
30030            int stremppolicy);
30031
```

30032 DESCRIPTION

30033 TRI The *posix_trace_attr_getinherited()* and *posix_trace_attr_setinherited()* functions, respectively, shall
 30034 get and set the inheritance policy stored in the *inheritance* attribute for traced processes across
 30035 the *fork()* and *spawn()* operations. The *inheritance* attribute of the attributes object pointed to by
 30036 the *attr* argument shall be set to one of the following values defined by manifest constants in the
 30037 <trace.h> header:

30038 POSIX_TRACE_CLOSE_FOR_CHILD

30039 After a *fork()* or *spawn()* operation, the child shall not be traced, and tracing of the parent
 30040 shall continue.

30041 POSIX_TRACE_INHERITED

30042 After a *fork()* or *spawn()* operation, if the parent is being traced, its child shall be
 30043 concurrently traced using the same trace stream.

30044 The default value for the *inheritance* attribute is POSIX_TRACE_CLOSE_FOR_CHILD.

30045 TRL The *posix_trace_attr_getlogfullpolicy()* and *posix_trace_attr_setlogfullpolicy()* functions,
 30046 respectively, shall get and set the trace log full policy stored in the *log-full-policy* attribute of the
 30047 attributes object pointed to by the *attr* argument.

30048 The *log-full-policy* attribute shall be set to one of the following values defined by manifest
 30049 constants in the <trace.h> header:

30050 POSIX_TRACE_LOOP

30051 The trace log shall loop until the associated trace stream is stopped. This policy means that
 30052 when the trace log gets full, the file system shall reuse the resources allocated to the oldest
 30053 trace events that were recorded. In this way, the trace log will always contain the most
 30054 recent trace events flushed.

30055 POSIX_TRACE_UNTIL_FULL

30056 The trace stream shall be flushed to the trace log until the trace log is full. This condition can
 30057 be deduced from the *posix_log_full_status* member status (see the **posix_trace_status_info**
 30058 structure defined in <trace.h>). The last recorded trace event shall be the

30059 POSIX_TRACE_STOP trace event.

30060 **POSIX_TRACE_APPEND**
 30061 The associated trace stream shall be flushed to the trace log without log size limitation. If
 30062 the application specifies **POSIX_TRACE_APPEND**, the implementation shall ignore the
 30063 `log-max-size` attribute.

30064 The default value for the `log-full-policy` attribute is **POSIX_TRACE_LOOP**.

30065 The `posix_trace_attr_getstreamfullpolicy()` and `posix_trace_attr_setstreamfullpolicy()` functions,
 30066 respectively, shall get and set the trace stream full policy stored in the `stream-full-policy` attribute
 30067 of the attributes object pointed to by the `attr` argument.

30068 The `stream-full-policy` attribute shall be set to one of the following values defined by manifest
 30069 constants in the `<trace.h>` header:

30070 **POSIX_TRACE_LOOP**
 30071 The trace stream shall loop until explicitly stopped by the `posix_trace_stop()` function. This
 30072 policy means that when the trace stream is full, the trace system shall reuse the resources
 30073 allocated to the oldest trace events recorded. In this way, the trace stream will always
 30074 contain the most recent trace events recorded.

30075 **POSIX_TRACE_UNTIL_FULL**
 30076 The trace stream will run until the trace stream resources are exhausted. Then the trace
 30077 stream will stop. This condition can be deduced from `posix_stream_status` and
 30078 `posix_stream_full_status` (see the **posix_trace_status_info** structure defined in `<trace.h>`).
 30079 When this trace stream is read, a **POSIX_TRACE_STOP** trace event shall be reported after
 30080 reporting the last recorded trace event. The trace system shall reuse the resources allocated
 30081 to any trace events already reported—see the `posix_trace_getnext_event()`,
 30082 `posix_trace_trygetnext_event()`, and `posix_trace_timedgetnext_event()` functions—or already
 30083 flushed for an active trace stream with log if the Trace Log option is supported; see the
 30084 `posix_trace_flush()` function. The trace system shall restart the trace stream when it is empty
 30085 and may restart it sooner. A **POSIX_TRACE_START** trace event shall be reported before
 30086 reporting the next recorded trace event.

30087 TRL **POSIX_TRACE_FLUSH**
 30088 If the Trace Log option is supported, this policy is identical to the
 30089 **POSIX_TRACE_UNTIL_FULL** trace stream full policy except that the trace stream shall be
 30090 flushed regularly as if `posix_trace_flush()` had been explicitly called. Defining this policy for
 30091 an active trace stream without log shall be invalid.

30092 The default value for the `stream-full-policy` attribute shall be **POSIX_TRACE_LOOP** for an active
 30093 trace stream without log.

30094 TRL If the Trace Log option is supported, the default value for the `stream-full-policy` attribute shall be
 30095 **POSIX_TRACE_FLUSH** for an active trace stream with log.

30096 **RETURN VALUE**

30097 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 30098 return the corresponding error number.

30099 TRI If successful, the `posix_trace_attr_getinherited()` function shall store the `inheritance` attribute value
 30100 in the object pointed to by `inheritancepolicy`. Otherwise, the content of this object is undefined.

30101 TRL If successful, the `posix_trace_attr_getlogfullpolicy()` function shall store the `log-full-policy` attribute
 30102 value in the object pointed to by `logpolicy`. Otherwise, the content of this object is undefined.

30103 If successful, the `posix_trace_attr_getstreamfullpolicy()` function shall store the `stream-full-policy`
 30104 attribute value in the object pointed to by `streampolicy`. Otherwise, the content of this object is

30105 undefined.

30106 ERRORS

30107 These functions may fail if:

30108 [EINVAL] The value specified by at least one of the arguments is invalid.

30109 EXAMPLES

30110 None.

30111 APPLICATION USAGE

30112 None.

30113 RATIONALE

30114 None.

30115 FUTURE DIRECTIONS

30116 None.

30117 SEE ALSO

30118 *fork()*, *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_flush()*, *posix_trace_get_attr()*,
30119 *posix_trace_getnext_event()*, *posix_trace_start()*, *posix_trace_timedgetnext_event()*, the Base
30120 Definitions volume of IEEE Std 1003.1-2001, <**trace.h**>

30121 CHANGE HISTORY

30122 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30123 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/39 is applied, adding the TRL and TRC 1
30124 margin codes to the *posix_trace_attr_setlogfullpolicy()* function. 1

30125 NAME

30126 posix_trace_attr_getlogsize, posix_trace_attr_getmaxdatasize,
30127 posix_trace_attr_getmaxsystemeventsizes, posix_trace_attr_getmaxuseventsizes,
30128 posix_trace_attr_getstreamsize, posix_trace_attr_setlogsize, posix_trace_attr_setmaxdatasize,
30129 posix_trace_attr_setstreamsize — retrieve and set trace stream size attributes (**TRACING**)

30130 SYNOPSIS

30152 DESCRIPTION

The `posix_trace_attr_getlogsize()` function shall copy the log size, in bytes, from the `log-max-size` attribute of the attributes object pointed to by the `attr` argument into the variable pointed to by the `logsize` argument. This log size is the maximum total of bytes that shall be allocated for system and user trace events in the trace log. The default value for the `log-max-size` attribute is implementation-defined.

30158 The `posix_trace_attr_setlogsize()` function shall set the maximum allowed size, in bytes, in the
30159 `log-max-size` attribute of the attributes object pointed to by the `attr` argument, using the size value
30160 supplied by the `logsize` argument.

30161 The trace log size shall be used if the *log-full-policy* attribute is set to POSIX_TRACE_LOOP or
30162 POSIX_TRACE_UNTIL_FULL. If the *log-full-policy* attribute is set to POSIX_TRACE_APPEND,
30163 the implementation shall ignore the *log-max-size* attribute.

30164 The *posix_trace_attr_getmaxdatasize()* function shall copy the maximum user trace event data
30165 size, in bytes, from the *max-data-size* attribute of the attributes object pointed to by the *attr*
30166 argument into the variable pointed to by the *maxdatasize* argument. The default value for the
30167 *max-data-size* attribute is implementation-defined.

30168 The `posix_trace_attr_getmaxsystemeventsiz()` function shall calculate the maximum memory size,
30169 in bytes, required to store a single system trace event. This value is calculated for the trace
30170 stream attributes object pointed to by the `attr` argument and is returned in the variable pointed
30171 to by the `eventsiz` argument.

30172 The values returned as the maximum memory sizes of the user and system trace events shall be
30173 such that if the sum of the maximum memory sizes of a set of the trace events that may be
30174 recorded in a trace stream is less than or equal to the *stream-min-size* attribute of that trace
30175 stream, the system provides the necessary resources for recording all those trace events, without
30176 loss.

30177 The *posix_trace_attr_getmaxusereventsizes()* function shall calculate the maximum memory size, in
30178 bytes, required to store a single user trace event generated by a call to *posix_trace_event()* with a
30179 *data_len* parameter equal to the *data_len* value specified in this call. This value is calculated for
30180 the trace stream attributes object pointed to by the *attr* argument and is returned in the variable
30181 pointed to by the *eventsizes* argument.

30182 The *posix_trace_attr_getstreamsize()* function shall copy the stream size, in bytes, from the
30183 *stream-min-size* attribute of the attributes object pointed to by the *attr* argument into the variable
30184 pointed to by the *streamsize* argument.

30185 This stream size is the current total memory size reserved for system and user trace events in the
30186 trace stream. The default value for the *stream-min-size* attribute is implementation-defined. The
30187 stream size refers to memory used to store trace event records. Other stream data (for example,
30188 trace attribute values) shall not be included in this size.

30189 The *posix_trace_attr_setmaxdatasize()* function shall set the maximum allowed size, in bytes, in
30190 the *max-data-size* attribute of the attributes object pointed to by the *attr* argument, using the size
30191 value supplied by the *maxdatasize* argument. This maximum size is the maximum allowed size
30192 for the user data argument which may be passed to *posix_trace_event()*. The implementation
30193 shall be allowed to truncate data passed to *trace_user_event* which is longer than *maxdatasize*.

30194 The *posix_trace_attr_setstreamsize()* function shall set the minimum allowed size, in bytes, in the
30195 *stream-min-size* attribute of the attributes object pointed to by the *attr* argument, using the size
30196 value supplied by the *streamsize* argument.

30197 RETURN VALUE

30198 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30199 return the corresponding error number.

30200 TRL The *posix_trace_attr_getlogsize()* function stores the maximum trace log allowed size in the object
30201 pointed to by *logsize*, if successful.

30202 The *posix_trace_attr_getmaxdatasize()* function stores the maximum trace event record memory
30203 size in the object pointed to by *maxdatasize*, if successful.

30204 The *posix_trace_attr_getmaxsystemeventsizes()* function stores the maximum memory size to store
30205 a single system trace event in the object pointed to by *eventsizes*, if successful.

30206 The *posix_trace_attr_getmaxusereventsizes()* function stores the maximum memory size to store a
30207 single user trace event in the object pointed to by *eventsizes*, if successful.

30208 The *posix_trace_attr_getstreamsize()* function stores the maximum trace stream allowed size in
30209 the object pointed to by *streamsize*, if successful.

30210 ERRORS

30211 These functions may fail if:

30212 [EINVAL] The value specified by one of the arguments is invalid.

30213 **EXAMPLES**

30214 None.

30215 **APPLICATION USAGE**

30216 None.

30217 **RATIONALE**

30218 None.

30219 **FUTURE DIRECTIONS**

30220 None.

30221 **SEE ALSO**

30222 *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_event()*, *posix_trace_get_attr()*, the Base
30223 Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <trace.h>

30224 **CHANGE HISTORY**

30225 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30226 NAME

30227 posix_trace_attr_getname — retrieve and set information about a trace stream (**TRACING**)

30228 SYNOPSIS

30229 TRC #include <trace.h>

30230 int posix_trace_attr_getname(const trace_attr_t *attr,
30231 char *tracename);

30232

30233 DESCRIPTION

30234 Refer to *posix_trace_attr_getclockres()*.

30235 NAME

30236 **posix_trace_attr_getstreamfullpolicy** — retrieve and set the behavior of a trace stream
30237 (**TRACING**)

30238 SYNOPSIS

```
30239 TRC #include <trace.h>
30240     int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *attr,
30241         int *stremppolicy);
30242
```

30243 DESCRIPTION

30244 Refer to *posix_trace_attr_getinherited()*.

30245 **NAME**30246 **posix_trace_attr_getstreamsize** — retrieve and set trace stream size attributes (**TRACING**)30247 **SYNOPSIS**

```
30248 TRC #include <sys/types.h>
30249 #include <trace.h>
30250 int posix_trace_attr_getstreamsize(const trace_attr_t *restrict attr,
30251         size_t *restrict streamsize);
30252
```

30253 **DESCRIPTION**30254 Refer to *posix_trace_attr_getlogsize()*.

30255 NAME

30256 **posix_trace_attr_init** — initialize the trace stream attributes object (**TRACING**)

30257 SYNOPSIS

30258 TRC `#include <trace.h>`

30259 `int posix_trace_attr_init(trace_attr_t *attr);`

30260

30261 DESCRIPTION

30262 Refer to *posix_trace_attr_destroy()*.

30263 **NAME**

30264 **posix_trace_attr_setinherited**, **posix_trace_attr_setlogfullpolicy** — retrieve and set the behavior
30265 of a trace stream (**TRACING**)

30266 **SYNOPSIS**

```
30267 TRC #include <trace.h>
30268 TRC TRI int posix_trace_attr_setinherited(trace_attr_t *attr,
30269           int inheritancepolicy);
30270 TRL int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
30271           int logpolicy);
30272
```

1

30273 **DESCRIPTION**

30274 Refer to *posix_trace_attr_getinherited()*.

30275 **NAME**

30276 **posix_trace_attr_setlogsize**, **posix_trace_attr_setmaxdatasize** — retrieve and set trace stream
30277 size attributes (**TRACING**)

30278 **SYNOPSIS**

```
30279 TRC #include <sys/types.h>
30280      #include <trace.h>

30281 TRC TRL int posix_trace_attr_setlogsize(trace_attr_t *attr,
30282           size_t logsize);
30283 TRC     int posix_trace_attr_setmaxdatasize(trace_attr_t *attr,
30284           size_t maxdatasize);
30285
```

30286 **DESCRIPTION**

30287 Refer to *posix_trace_attr_getlogsize()*.

30288 NAME

30289 **posix_trace_attr_setname** — retrieve and set information about a trace stream (**TRACING**)

30290 SYNOPSIS

30291 TRC

```
#include <trace.h>
```

30292

```
int posix_trace_attr_setname(trace_attr_t *attr,
```

30293

```
const char *tracename);
```

30294

30295 DESCRIPTION

30296 Refer to *posix_trace_attr_getclockres()*.

30297 **NAME**30298 **posix_trace_attr_setstreamfullpolicy** — retrieve and set the behavior of a trace stream
30299 (**TRACING**)30300 **SYNOPSIS**30301 TRC

```
#include <trace.h>
```


30302 int posix_trace_attr_setstreamfullpolicy(trace_attr_t *attr,
30303 int strempolicy);
30304

1

30305 **DESCRIPTION**30306 Refer to *posix_trace_attr_getinherited()*.

30307 **NAME**30308 **posix_trace_attr_setstreamsize** — retrieve and set trace stream size attributes (**TRACING**)30309 **SYNOPSIS**

```
30310 TRC #include <sys/types.h>
30311 #include <trace.h>
30312 int posix_trace_attr_setstreamsize(trace_attr_t *attr,
30313         size_t streamsiz);
30314
```

30315 **DESCRIPTION**30316 Refer to *posix_trace_attr_getlogsize()*.

30317 NAME

30318 **posix_trace_clear** — clear trace stream and trace log (**TRACING**)

30319 SYNOPSIS

```
30320 TRC #include <sys/types.h>
30321 #include <trace.h>
30322 int posix_trace_clear(trace_id_t trid);
30323
```

30324 DESCRIPTION

30325 The *posix_trace_clear()* function shall reinitialize the trace stream identified by the argument *trid*
30326 as if it were returning from the *posix_trace_create()* function, except that the same allocated
30327 resources shall be reused, the mapping of trace event type identifiers to trace event names shall
30328 be unchanged, and the trace stream status shall remain unchanged (that is, if it was running, it
30329 remains running and if it was suspended, it remains suspended).

30330 All trace events in the trace stream recorded before the call to *posix_trace_clear()* shall be lost.
30331 The *posix_stream_full_status* status shall be set to **POSIX_TRACE_NOT_FULL**. There is no
30332 guarantee that all trace events that occurred during the *posix_trace_clear()* call are recorded; the
30333 behavior with respect to trace points that may occur during this call is unspecified.

30334 TRL If the Trace Log option is supported and the trace stream has been created with a log, the
30335 *posix_trace_clear()* function shall reinitialize the trace stream with the same behavior as if the
30336 trace stream was created without the log, plus it shall reinitialize the trace log associated with
30337 the trace stream identified by the argument *trid* as if it were returning from the
30338 *posix_trace_create_withlog()* function, except that the same allocated resources, for the trace log,
30339 may be reused and the associated trace stream status remains unchanged. The first trace event
30340 recorded in the trace log after the call to *posix_trace_clear()* shall be the same as the first trace
30341 event recorded in the active trace stream after the call to *posix_trace_clear()*. The
30342 *posix_log_full_status* status shall be set to **POSIX_TRACE_NOT_FULL**. There is no guarantee that
30343 all trace events that occurred during the *posix_trace_clear()* call are recorded in the trace log; the
30344 behavior with respect to trace points that may occur during this call is unspecified. If the log full
30345 policy is **POSIX_TRACE_APPEND**, the effect of a call to this function is unspecified for the trace
30346 log associated with the trace stream identified by the *trid* argument.

30347 RETURN VALUE

30348 Upon successful completion, the *posix_trace_clear()* function shall return a value of zero.
30349 Otherwise, it shall return the corresponding error number.

30350 ERRORS

30351 The *posix_trace_clear()* function shall fail if:

30352 [EINVAL] The value of the *trid* argument does not correspond to an active trace stream.

30353 EXAMPLES

30354 None.

30355 APPLICATION USAGE

30356 None.

30357 RATIONALE

30358 None.

30359 FUTURE DIRECTIONS

30360 None.

30361 **SEE ALSO**

30362 *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_flush()*, *posix_trace_get_attr()*, the Base
30363 Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <trace.h>

30364 **CHANGE HISTORY**

30365 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30366 IEEE PASC Interpretation 1003.1 #123 is applied.

30367 NAME

30368 posix_trace_close, posix_trace_open, posix_trace_rewind — trace log management (**TRACING**)

30369 SYNOPSIS

30370 TRC TRL #include <trace.h>

```
30371     int posix_trace_close(trace_id_t trid);
30372     int posix_trace_open(int file_desc, trace_id_t *trid);
30373     int posix_trace_rewind(trace_id_t trid);
```

30375 DESCRIPTION

30376 The *posix_trace_close()* function shall deallocate the trace log identifier indicated by *trid*, and all
 30377 of its associated resources. If there is no valid trace log pointed to by the *trid*, this function shall
 30378 fail.

30379 The *posix_trace_open()* function shall allocate the necessary resources and establish the
 30380 connection between a trace log identified by the *file_desc* argument and a trace stream identifier
 30381 identified by the object pointed to by the *trid* argument. The *file_desc* argument should be a valid
 30382 open file descriptor that corresponds to a trace log. The *file_desc* argument shall be open for
 30383 reading. The current trace event timestamp, which specifies the timestamp of the trace event
 30384 that will be read by the next call to *posix_trace_getnext_event()*, shall be set to the timestamp of
 30385 the oldest trace event recorded in the trace log identified by *trid*.

30386 The *posix_trace_open()* function shall return a trace stream identifier in the variable pointed to by
 30387 the *trid* argument, that may only be used by the following functions:

30388 <i>posix_trace_close()</i>	30388 <i>posix_trace_get_attr()</i>
30389 <i>posix_trace_eventid_equal()</i>	30389 <i>posix_trace_get_status()</i>
30390 <i>posix_trace_eventid_get_name()</i>	30390 <i>posix_trace_getnext_event()</i>
30391 <i>posix_trace_eventtypelist_getnext_id()</i>	30391 <i>posix_trace_rewind()</i>
30392 <i>posix_trace_eventtypelist_rewind()</i>	

30393 In particular, notice that the operations normally used by a trace controller process, such as
 30394 *posix_trace_start()*, *posix_trace_stop()*, or *posix_trace_shutdown()*, cannot be invoked using the
 30395 trace stream identifier returned by the *posix_trace_open()* function.

30396 The *posix_trace_rewind()* function shall reset the current trace event timestamp, which specifies
 30397 the timestamp of the trace event that will be read by the next call to *posix_trace_getnext_event()*,
 30398 to the timestamp of the oldest trace event recorded in the trace log identified by *trid*.

30399 RETURN VALUE

30400 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 30401 return the corresponding error number.

30402 If successful, the *posix_trace_open()* function stores the trace stream identifier value in the object
 30403 pointed to by *trid*.

30404 ERRORS

30405 The *posix_trace_open()* function shall fail if:

30406 [EINTR] The operation was interrupted by a signal and thus no trace log was opened.

30407 [EINVAL] The object pointed to by *file_desc* does not correspond to a valid trace log.

30408 The *posix_trace_close()* and *posix_trace_rewind()* functions may fail if:

30409 [EINVAL] The object pointed to by *trid* does not correspond to a valid trace log.

30410 EXAMPLES

30411 None.

30412 APPLICATION USAGE

30413 None.

30414 RATIONALE

30415 None.

30416 FUTURE DIRECTIONS

30417 None.

30418 SEE ALSO

30419 *posix_trace_get_attr()*, *posix_trace_get_filter()*, *posix_trace_getnext_event()*, the Base Definitions

30420 volume of IEEE Std 1003.1-2001, <trace.h>

30421 CHANGE HISTORY

30422 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30423 IEEE PASC Interpretation 1003.1 #123 is applied.

30424 NAME

30425 posix_trace_create, posix_trace_create_withlog, posix_trace_flush, posix_trace_shutdown —
 30426 trace stream initialization, flush, and shutdown from a process (TRACING)

30427 SYNOPSIS

```
30428 TRC #include <sys/types.h>
30429 #include <trace.h>

30430     int posix_trace_create(pid_t pid,
30431         const trace_attr_t *restrict attr,
30432         trace_id_t *restrict trid);
30433 TRL int posix_trace_create_withlog(pid_t pid,
30434         const trace_attr_t *restrict attr, int file_desc,
30435         trace_id_t *restrict trid);
30436     int posix_trace_flush(trace_id_t trid);
30437 TRC int posix_trace_shutdown(trace_id_t trid);
30438
```

30439 DESCRIPTION

30440 The *posix_trace_create()* function shall create an active trace stream. It allocates all the resources
 30441 needed by the trace stream being created for tracing the process specified by *pid* in accordance
 30442 with the *attr* argument. The *attr* argument represents the initial attributes of the trace stream and
 30443 shall have been initialized by the function *posix_trace_attr_init()* prior to the *posix_trace_create()*
 30444 call. If the argument *attr* is NULL, the default attributes shall be used. The *attr* attributes object
 30445 shall be manipulated through a set of functions described in the *posix_trace_attr* family of
 30446 functions. If the attributes of the object pointed to by *attr* are modified later, the attributes of the
 30447 trace stream shall not be affected. The *creation-time* attribute of the newly created trace stream
 30448 shall be set to the value of the system clock, if the Timers option is not supported, or to the value
 30449 of the CLOCK_REALTIME clock, if the Timers option is supported.

30450 The *pid* argument represents the target process to be traced. If the process executing this
 30451 function does not have appropriate privileges to trace the process identified by *pid*, an error shall
 30452 be returned. If the *pid* argument is zero, the calling process shall be traced.

30453 The *posix_trace_create()* function shall store the trace stream identifier of the new trace stream in
 30454 the object pointed to by the *trid* argument. This trace stream identifier shall be used in
 30455 subsequent calls to control tracing. The *trid* argument may only be used by the following
 30456 functions:

30457 <i>posix_trace_clear()</i>	30457 <i>posix_trace_getnext_event()</i>
30458 <i>posix_trace_eventid_equal()</i>	30458 <i>posix_trace_shutdown()</i>
30459 <i>posix_trace_eventid_get_name()</i>	30459 <i>posix_trace_start()</i>
30460 <i>posix_trace_eventtypelist_getnext_id()</i>	30460 <i>posix_trace_stop()</i>
30461 <i>posix_trace_eventtypelist_rewind()</i>	30461 <i>posix_trace_timedgetnext_event()</i>
30462 <i>posix_trace_get_attr()</i>	30462 <i>posix_trace_trid_eventid_open()</i>
30463 <i>posix_trace_get_status()</i>	30463 <i>posix_trace_trygetnext_event()</i>

30464 TEF If the Trace Event Filter option is supported, the following additional functions may use the *trid*
 30465 argument:

30466 *posix_trace_get_filter()* *posix_trace_set_filter()*

30467

30468 In particular, notice that the operations normally used by a trace analyzer process, such as
 30469 *posix_trace_rewind()* or *posix_trace_close()*, cannot be invoked using the trace stream identifier
 30470 returned by the *posix_trace_create()* function.

30471 TEF A trace stream shall be created in a suspended state. If the Trace Event Filter option is
 30472 supported, its trace event type filter shall be empty.

30473 The *posix_trace_create()* function may be called multiple times from the same or different
 30474 processes, with the system-wide limit indicated by the runtime invariant value
 30475 {TRACE_SYS_MAX}, which has the minimum value {_POSIX_TRACE_SYS_MAX}.

30476 The trace stream identifier returned by the *posix_trace_create()* function in the argument pointed
 30477 to by *trid* is valid only in the process that made the function call. If it is used from another
 30478 process, that is a child process, in functions defined in IEEE Std 1003.1-2001, these functions shall
 30479 return with the error [EINVAL].

30480 TRL The *posix_trace_create_withlog()* function shall be equivalent to *posix_trace_create()*, except that it
 30481 associates a trace log with this stream. The *file_desc* argument shall be the file descriptor
 30482 designating the trace log destination. The function shall fail if this file descriptor refers to a file
 30483 with a file type that is not compatible with the log policy associated with the trace log. The list of
 30484 the appropriate file types that are compatible with each log policy is implementation-defined.

30485 The *posix_trace_create_withlog()* function shall return in the parameter pointed to by *trid* the trace
 30486 stream identifier, which uniquely identifies the newly created trace stream, and shall be used in
 30487 subsequent calls to control tracing. The *trid* argument may only be used by the following
 30488 functions:

<i>posix_trace_clear()</i>	<i>posix_trace_getnext_event()</i>
<i>posix_trace_eventid_equal()</i>	<i>posix_trace_shutdown()</i>
<i>posix_trace_eventid_get_name()</i>	<i>posix_trace_start()</i>
<i>posix_trace_eventtypelist_getnext_id()</i>	<i>posix_trace_stop()</i>
<i>posix_trace_eventtypelist_rewind()</i>	<i>posix_trace_timedgetnext_event()</i>
<i>posix_trace_flush()</i>	<i>posix_trace_trid_eventid_open()</i>
<i>posix_trace_get_attr()</i>	<i>posix_trace_trygetnext_event()</i>
<i>posix_trace_get_status()</i>	

30497

30498 TRL TEF If the Trace Event Filter option is supported, the following additional functions may use the *trid*
 30499 argument:

30500 *posix_trace_get_filter()* *posix_trace_set_filter()*

30501

30502 TRL In particular, notice that the operations normally used by a trace analyzer process, such as
 30503 *posix_trace_rewind()* or *posix_trace_close()*, cannot be invoked using the trace stream identifier
 30504 returned by the *posix_trace_create_withlog()* function.

30505 The *posix_trace_flush()* function shall initiate a flush operation which copies the contents of the
 30506 trace stream identified by the argument *trid* into the trace log associated with the trace stream at
 30507 the creation time. If no trace log has been associated with the trace stream pointed to by *trid*, this
 30508 function shall return an error. The termination of the flush operation can be polled by the
 30509 *posix_trace_get_status()* function. During the flush operation, it shall be possible to trace new
 30510 trace events up to the point when the trace stream becomes full. After flushing is completed, the
 30511 space used by the flushed trace events shall be available for tracing new trace events.

30512 If flushing the trace stream causes the resulting trace log to become full, the trace log full policy
 30513 shall be applied. If the trace *log-full-policy* attribute is set, the following occurs:

30514 **POSIX_TRACE_UNTIL_FULL**
 30515 The trace events that have not yet been flushed shall be discarded.

30516 **POSIX_TRACE_LOOP**
 30517 The trace events that have not yet been flushed shall be written to the beginning of the trace
 30518 log, overwriting previous trace events stored there.

30519 **POSIX_TRACE_APPEND**
 30520 The trace events that have not yet been flushed shall be appended to the trace log.

30521

30522 The *posix_trace_shutdown()* function shall stop the tracing of trace events in the trace stream
 30523 identified by *trid*, as if *posix_trace_stop()* had been invoked. The *posix_trace_shutdown()* function
 30524 shall free all the resources associated with the trace stream.

30525 The *posix_trace_shutdown()* function shall not return until all the resources associated with the
 30526 trace stream have been freed. When the *posix_trace_shutdown()* function returns, the *trid*
 30527 argument becomes an invalid trace stream identifier. A call to this function shall unconditionally
 30528 deallocate the resources regardless of whether all trace events have been retrieved by the
 30529 analyzer process. Any thread blocked on one of the *trace_getnext_event()* functions (which
 30530 specified this *trid*) before this call is unblocked with the error [EINVAL].

30531 If the process exits, invokes a member of the *exec* family of functions, or is terminated, the trace
 30532 streams that the process had created and that have not yet been shut down, shall be
 30533 automatically shut down as if an explicit call were made to the *posix_trace_shutdown()* function.

30534 TRL For an active trace stream with log, when the *posix_trace_shutdown()* function is called, all trace
 30535 events that have not yet been flushed to the trace log shall be flushed, as in the
 30536 *posix_trace_flush()* function, and the trace log shall be closed.

30537 When a trace log is closed, all the information that may be retrieved later from the trace log
 30538 through the trace interface shall have been written to the trace log. This information includes the
 30539 trace attributes, the list of trace event types (with the mapping between trace event names and
 30540 trace event type identifiers), and the trace status.

30541 In addition, unspecified information shall be written to the trace log to allow detection of a valid
 30542 trace log during the *posix_trace_open()* operation.

30543 The *posix_trace_shutdown()* function shall not return until all trace events have been flushed.

30544 **RETURN VALUE**

30545 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 30546 return the corresponding error number.

30547 TRL The *posix_trace_create()* and *posix_trace_create_withlog()* functions store the trace stream
 30548 identifier value in the object pointed to by *trid*, if successful.

30549 **ERRORS**

30550 TRL The *posix_trace_create()* and *posix_trace_create_withlog()* functions shall fail if:

30551 [EAGAIN]	No more trace streams can be started now. {TRACE_SYS_MAX} has been exceeded.
30553 [EINTR]	The operation was interrupted by a signal. No trace stream was created.
30554 [EINVAL]	One or more of the trace parameters specified by the <i>attr</i> parameter is invalid.

30555	[ENOMEM]	The implementation does not currently have sufficient memory to create the trace stream with the specified parameters.
30556		
30557	[EPERM]	The caller does not have appropriate privilege to trace the process specified by <i>pid</i> .
30558		
30559	[ESRCH]	The <i>pid</i> argument does not refer to an existing process.
30560 TRL	The <i>posix_trace_create_withlog()</i> function shall fail if:	
30561	[EBADF]	The <i>file_desc</i> argument is not a valid file descriptor open for writing.
30562	[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.
30563		
30564	[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.
30565		
30566		
30567 TRL	The <i>posix_trace_flush()</i> and <i>posix_trace_shutdown()</i> functions shall fail if:	
30568	[EINVAL]	The value of the <i>trid</i> argument does not correspond to an active trace stream with log.
30569		
30570	[EFBIG]	The trace log file has attempted to exceed an implementation-defined maximum file size.
30571		
30572	[ENOSPC]	No space left on device.
30573		

30574 EXAMPLES

30575 None.

30576 APPLICATION USAGE

30577 None.

30578 RATIONALE

30579 None.

30580 FUTURE DIRECTIONS

30581 None.

30582 SEE ALSO

30583 *clock_getres()*, *exec*, *posix_trace_attr_init()*, *posix_trace_clear()*, *posix_trace_close()*,
30584 *posix_trace_eventid_equal()*, *posix_trace_eventtypelist_getnext_id()*, *posix_trace_flush()*,
30585 *posix_trace_get_attr()*, *posix_trace_get_filter()*, *posix_trace_get_status()*, *posix_trace_getnext_event()*,
30586 *posix_trace_open()*, *posix_trace_set_filter()*, *posix_trace_shutdown()*, *posix_trace_start()*,
30587 *posix_trace_timedgetnext_event()*, *posix_trace_trid_eventid_open()*, *posix_trace_start()*, *time()*, the
30588 Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <trace.h>

30589 CHANGE HISTORY

30590 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30591 NAME

30592 posix_trace_event, posix_trace_eventid_open — trace functions for instrumenting application
 30593 code (**TRACING**)

30594 SYNOPSIS

```
30595 TRC #include <sys/types.h>
30596 #include <trace.h>

30597 void posix_trace_event(trace_event_id_t event_id,
30598     const void *restrict data_ptr, size_t data_len);
30599 int posix_trace_eventid_open(const char *restrict event_name,
30600     trace_event_id_t *restrict event_id);
```

30601

30602 DESCRIPTION

30603 The *posix_trace_event()* function shall record the *event_id* and the user data pointed to by *data_ptr*
 30604 in the trace stream into which the calling process is being traced and in which *event_id* is not
 30605 filtered out. If the total size of the user trace event data represented by *data_len* is not greater
 30606 than the declared maximum size for user trace event data, then the *truncation-status* attribute of
 30607 the trace event recorded is **POSIX_TRACE_NOT_TRUNCATED**. Otherwise, the user trace event
 30608 data is truncated to this declared maximum size and the *truncation-status* attribute of the trace
 30609 event recorded is **POSIX_TRACE_TRUNCATED_RECORD**.

30610 If there is no trace stream created for the process or if the created trace stream is not running, or
 30611 if the trace event specified by *event_id* is filtered out in the trace stream, the *posix_trace_event()*
 30612 function shall have no effect.

30613 The *posix_trace_eventid_open()* function shall associate a user trace event name with a trace event
 30614 type identifier for the calling process. The trace event name is the string pointed to by the
 30615 argument *event_name*. It shall have a maximum of {TRACE_EVENT_NAME_MAX} characters
 30616 (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). The number of user
 30617 trace event type identifiers that can be defined for any given process is limited by the maximum
 30618 value {TRACE_USER_EVENT_MAX}, which has the minimum value
 30619 {POSIX_TRACE_USER_EVENT_MAX}.

30620 If the Trace Inherit option is not supported, the *posix_trace_eventid_open()* function shall
 30621 associate the user trace event name pointed to by the *event_name* argument with a trace event type identifier
 30622 that is unique for the traced process, and is returned in the variable pointed to by
 30623 the *event_id* argument. If the user trace event name has already been mapped for the traced
 30624 process, then the previously assigned trace event type identifier shall be returned. If the per-
 30625 process user trace event name limit represented by {TRACE_USER_EVENT_MAX} has been
 30626 reached, the pre-defined **POSIX_TRACE_UNNAMED_USEREVENT** (see Table 2-7 (on page 80))
 30627 user trace event shall be returned.

30628 TRI If the Trace Inherit option is supported, the *posix_trace_eventid_open()* function shall associate the
 30629 user trace event name pointed to by the *event_name* argument with a trace event type identifier
 30630 that is unique for all the processes being traced in this same trace stream, and is returned in the
 30631 variable pointed to by the *event_id* argument. If the user trace event name has already been
 30632 mapped for the traced processes, then the previously assigned trace event type identifier shall be
 30633 returned. If the per-process user trace event name limit represented by
 30634 {TRACE_USER_EVENT_MAX} has been reached, the pre-defined
 30635 **POSIX_TRACE_UNNAMED_USEREVENT** (Table 2-7 (on page 80)) user trace event shall be
 30636 returned.
 30637 **Note:** The above procedure, together with the fact that multiple processes can only be traced into the
 30638 same trace stream by inheritance, ensure that all the processes that are traced into a trace
 30639 stream have the same mapping of trace event names to trace event type identifiers.

30640

30641 If there is no trace stream created, the *posix_trace_eventid_open()* function shall store this
30642 information for future trace streams created for this process.

30643 RETURN VALUE

30644 No return value is defined for the *posix_trace_event()* function.

30645 Upon successful completion, the *posix_trace_eventid_open()* function shall return a value of zero.
30646 Otherwise, it shall return the corresponding error number. The *posix_trace_eventid_open()*
30647 function stores the trace event type identifier value in the object pointed to by *event_id*, if
30648 successful.

30649 ERRORS

30650 The *posix_trace_eventid_open()* function shall fail if:

30651 [ENAMETOOLONG]

30652 The size of the name pointed to by the *event_name* argument was longer than
30653 the implementation-defined value {TRACE_EVENT_NAME_MAX}.

30654 EXAMPLES

30655 None.

30656 APPLICATION USAGE

30657 None.

30658 RATIONALE

30659 None.

30660 FUTURE DIRECTIONS

30661 None.

30662 SEE ALSO

30663 Table 2-7 (on page 80), *posix_trace_start()*, *posix_trace_trid_eventid_open()*, the Base Definitions
30664 volume of IEEE Std 1003.1-2001, <sys/types.h>, <trace.h>

30665 CHANGE HISTORY

30666 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30667 IEEE PASC Interpretation 1003.1 #123 is applied.

30668 IEEE PASC Interpretation 1003.1 #127 is applied, correcting some editorial errors in the names of
30669 the *posix_trace_eventid_open()* function and the *event_id* argument.

30670 NAME

30671 posix_trace_eventid_equal, posix_trace_eventid_get_name, posix_trace_trid_eventid_open —
 30672 manipulate the trace event type identifier (**TRACING**)

30673 SYNOPSIS

```
30674 TRC #include <trace.h>
30675
30676     int posix_trace_eventid_equal(trace_id_t trid, trace_event_id_t event1,
30677                                     trace_event_id_t event2);
30678     int posix_trace_eventid_get_name(trace_id_t trid,
30679                                     trace_event_id_t event, char *event_name);
30680 TEF     int posix_trace_trid_eventid_open(trace_id_t trid,
30681                                         const char *restrict event_name,
30682                                         trace_event_id_t *restrict event);
```

30683 DESCRIPTION

30684 The *posix_trace_eventid_equal()* function shall compare the trace event type identifiers *event1* and
 30685 *event2* from the same trace stream or the same trace log identified by the *trid* argument. If the
 30686 trace event type identifiers *event1* and *event2* are from different trace streams, the return value
 30687 shall be unspecified.

30688 The *posix_trace_eventid_get_name()* function shall return, in the argument pointed to by
 30689 *event_name*, the trace event name associated with the trace event type identifier identified by the
 30690 argument *event*, for the trace stream or for the trace log identified by the *trid* argument. The
 30691 name of the trace event shall have a maximum of {TRACE_EVENT_NAME_MAX} characters
 30692 (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). Successive calls to
 30693 this function with the same trace event type identifier and the same trace stream identifier shall
 30694 return the same event name.

30695 TEF The *posix_trace_trid_eventid_open()* function shall associate a user trace event name with a trace
 30696 event type identifier for a given trace stream. The trace stream is identified by the *trid* argument,
 30697 and it shall be an active trace stream. The trace event name is the string pointed to by the
 30698 argument *event_name*. It shall have a maximum of {TRACE_EVENT_NAME_MAX} characters
 30699 (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). The number of user
 30700 trace event type identifiers that can be defined for any given process is limited by the maximum
 30701 value {TRACE_USER_EVENT_MAX}, which has the minimum value
 30702 {_POSIX_TRACE_USER_EVENT_MAX}.

30703 If the Trace Inherit option is not supported, the *posix_trace_trid_eventid_open()* function shall
 30704 associate the user trace event name pointed to by the *event_name* argument with a trace event
 30705 type identifier that is unique for the process being traced in the trace stream identified by the *trid*
 30706 argument, and is returned in the variable pointed to by the *event* argument. If the user trace
 30707 event name has already been mapped for the traced process, then the previously assigned trace
 30708 event type identifier shall be returned. If the per-process user trace event name limit represented
 30709 by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined
 30710 POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 80)) user trace event shall be
 30711 returned.

30712 TEF TRI If the Trace Inherit option is supported, the *posix_trace_trid_eventid_open()* function shall
 30713 associate the user trace event name pointed to by the *event_name* argument with a trace event
 30714 type identifier that is unique for all the processes being traced in the trace stream identified by
 30715 the *trid* argument, and is returned in the variable pointed to by the *event* argument. If the user trace
 30716 event name has already been mapped for the traced processes, then the previously assigned trace
 30717 event type identifier shall be returned. If the per-process user trace event name limit represented
 30718 by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined

30719 POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 80)) user trace event shall be
30720 returned.

30721 RETURN VALUE

30722 TEF Upon successful completion, the *posix_trace_eventid_get_name()* and
30723 *posix_trace_trid_eventid_open()* functions shall return a value of zero. Otherwise, they shall return
30724 the corresponding error number.

30725 The *posix_trace_eventid_equal()* function shall return a non-zero value if *event1* and *event2* are
30726 equal; otherwise, a value of zero shall be returned. No errors are defined. If either *event1* or
30727 *event2* are not valid trace event type identifiers for the trace stream specified by *trid* or if the *trid*
30728 is invalid, the behavior shall be unspecified.

30729 The *posix_trace_eventid_get_name()* function stores the trace event name value in the object
30730 pointed to by *event_name*, if successful.

30731 TEF The *posix_trace_trid_eventid_open()* function stores the trace event type identifier value in the
30732 object pointed to by *event*, if successful.

30733 ERRORS

30734 TEF The *posix_trace_eventid_get_name()* and *posix_trace_trid_eventid_open()* functions shall fail if:
30735 [EINVAL] The *trid* argument was not a valid trace stream identifier.

30736 TEF The *posix_trace_trid_eventid_open()* function shall fail if:

30737 TEF [ENAMETOOLONG]
30738 The size of the name pointed to by the *event_name* argument was longer than
30739 the implementation-defined value {TRACE_EVENT_NAME_MAX}.

30740 The *posix_trace_eventid_get_name()* function shall fail if:

30741 [EINVAL] The trace event type identifier *event* was not associated with any name.

30742 EXAMPLES

30743 None.

30744 APPLICATION USAGE

30745 None.

30746 RATIONALE

30747 None.

30748 FUTURE DIRECTIONS

30749 None.

30750 SEE ALSO

30751 Table 2-7 (on page 80), *posix_trace_event()*, *posix_trace_getnext_event()*, the Base Definitions
30752 volume of IEEE Std 1003.1-2001, <*trace.h*>

30753 CHANGE HISTORY

30754 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30755 IEEE PASC Interpretations 1003.1 #123 and #129 are applied.

30756 NAME

30757 **posix_trace_eventid_open** — trace functions for instrumenting application code (**TRACING**)

30758 SYNOPSIS

```
30759 TRC #include <sys/types.h>
30760      #include <trace.h>
30761      int posix_trace_eventid_open(const char *restrict event_name,
30762                                     trace_event_id_t *restrict event_id);
30763
```

30764 DESCRIPTION

30765 Refer to *posix_trace_event()*.

30766 NAME

30767 posix_trace_eventset_add, posix_trace_eventset_del, posix_trace_eventset_empty,
 30768 posix_trace_eventset_fill, posix_trace_eventset_ismember — manipulate trace event type sets
 30769 (TRACING)

30770 SYNOPSIS

```
30771 TRC TEF #include <trace.h>
30772
30773     int posix_trace_eventset_add(trace_event_id_t event_id,
30774         trace_event_set_t *set);
30775     int posix_trace_eventset_del(trace_event_id_t event_id,
30776         trace_event_set_t *set);
30777     int posix_trace_eventset_empty(trace_event_set_t *set);
30778     int posix_trace_eventset_fill(trace_event_set_t *set, int what);
30779     int posix_trace_eventset_ismember(trace_event_id_t event_id,
30780         const trace_event_set_t *restrict set, int *restrict ismember);
```

30781 DESCRIPTION

30782 These primitives manipulate sets of trace event types. They operate on data objects addressable by the application, not on the current trace event filter of any trace stream.

30784 The *posix_trace_eventset_add()* and *posix_trace_eventset_del()* functions, respectively, shall add or delete the individual trace event type specified by the value of the argument *event_id* to or from the trace event type set pointed to by the argument *set*. Adding a trace event type already in the set or deleting a trace event type not in the set shall not be considered an error.

30788 The *posix_trace_eventset_empty()* function shall initialize the trace event type set pointed to by the *set* argument such that all trace event types defined, both system and user, shall be excluded from the set.

30791 The *posix_trace_eventset_fill()* function shall initialize the trace event type set pointed to by the argument *set*, such that the set of trace event types defined by the argument *what* shall be included in the set. The value of the argument *what* shall consist of one of the following values, as defined in the <trace.h> header:

30795 POSIX_TRACE_WOPID_EVENTS

30796 All the process-independent implementation-defined system trace event types are included in the set.

30798 POSIX_TRACE_SYSTEM_EVENTS

30799 All the implementation-defined system trace event types are included in the set, as are those defined in IEEE Std 1003.1-2001.

30801 POSIX_TRACE_ALL_EVENTS

30802 All trace event types defined, both system and user, are included in the set.

30803 Applications shall call either *posix_trace_eventset_empty()* or *posix_trace_eventset_fill()* at least once for each object of type **trace_event_set_t** prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of the *posix_trace_eventset_add()*, *posix_trace_eventset_del()*, or *posix_trace_eventset_ismember()* functions, the results are undefined.

30808 The *posix_trace_eventset_ismember()* function shall test whether the trace event type specified by the value of the argument *event_id* is a member of the set pointed to by the argument *set*. The value returned in the object pointed to by *ismember* argument is zero if the trace event type identifier is not a member of the set and a value different from zero if it is a member of the set.

30812 RETURN VALUE

30813 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30814 return the corresponding error number.

30815 ERRORS

30816 These functions may fail if:

30817 [EINVAL] The value of one of the arguments is invalid.

30818 EXAMPLES

30819 None.

30820 APPLICATION USAGE

30821 None.

30822 RATIONALE

30823 None.

30824 FUTURE DIRECTIONS

30825 None.

30826 SEE ALSO

30827 *posix_trace_set_filter()*, *posix_trace_trid_eventid_open()*, the Base Definitions volume of
30828 IEEE Std 1003.1-2001, <trace.h>

30829 CHANGE HISTORY

30830 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30831 NAME

30832 **posix_trace_eventtypelist_getnext_id**, **posix_trace_eventtypelist_rewind** — iterate over a
30833 mapping of trace event types (**TRACING**)

30834 SYNOPSIS

```
30835 TRC #include <trace.h>
30836     int posix_trace_eventtypelist_getnext_id(trace_id_t trid,
30837           trace_event_id_t *restrict event, int *restrict unavailable);
30838     int posix_trace_eventtypelist_rewind(trace_id_t trid);
30839
```

30840 DESCRIPTION

30841 The first time *posix_trace_eventtypelist_getnext_id()* is called, the function shall return in the
30842 variable pointed to by *event* the first trace event type identifier of the list of trace events of the
30843 trace stream identified by the *trid* argument. Successive calls to
30844 *posix_trace_eventtypelist_getnext_id()* return in the variable pointed to by *event* the next trace
30845 event type identifier in that same list. Each time a trace event type identifier is successfully
30846 written into the variable pointed to by the *event* argument, the variable pointed to by the
30847 *unavailable* argument shall be set to zero. When no more trace event type identifiers are
30848 available, and so none is returned, the variable pointed to by the *unavailable* argument shall be
30849 set to a value different from zero.

30850 The *posix_trace_eventtypelist_rewind()* function shall reset the next trace event type identifier to
30851 be read to the first trace event type identifier from the list of trace events used in the trace stream
30852 identified by *trid*.

30853 RETURN VALUE

30854 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
30855 return the corresponding error number.

30856 The *posix_trace_eventtypelist_getnext_id()* function stores the trace event type identifier value in
30857 the object pointed to by *event*, if successful.

30858 ERRORS

30859 These functions shall fail if:

30860 [EINVAL] The *trid* argument was not a valid trace stream identifier.

30861 EXAMPLES

30862 None.

30863 APPLICATION USAGE

30864 None.

30865 RATIONALE

30866 None.

30867 FUTURE DIRECTIONS

30868 None.

30869 SEE ALSO

30870 *posix_trace_event()*, *posix_trace_getnext_event()*, *posix_trace_trid_eventid_open()*, the Base
30871 Definitions volume of IEEE Std 1003.1-2001, <**trace.h**>

30872 CHANGE HISTORY

30873 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30874 IEEE PASC Interpretations 1003.1 #123 and #129 are applied.

30875 **NAME**30876 **posix_trace_flush** — trace stream flush from a process (**TRACING**)30877 **SYNOPSIS**30878 TRC #include <sys/types.h>
30879 #include <trace.h>30880 TRC TRL int posix_trace_flush(trace_id_t *trid*);

30881

1

30882 **DESCRIPTION**30883 Refer to *posix_trace_create()*.

30884 NAME

30885 **posix_trace_get_attr**, **posix_trace_get_status** — retrieve the trace attributes or trace status
 30886 **(TRACING)**

30887 SYNOPSIS

```
30888 TRC #include <trace.h>
30889     int posix_trace_get_attr(trace_id_t trid, trace_attr_t *attr);
30890     int posix_trace_get_status(trace_id_t trid,
30891         struct posix_trace_status_info *statusinfo);
30892
```

30893 DESCRIPTION

30894 The *posix_trace_get_attr()* function shall copy the attributes of the active trace stream identified
 30895 by *trid* into the object pointed to by the *attr* argument. If the Trace Log option is supported, *trid*
 30896 may represent a pre-recorded trace log.

30897 The *posix_trace_get_status()* function shall return, in the structure pointed to by the *statusinfo*
 30898 argument, the current trace status for the trace stream identified by the *trid* argument. These
 30899 status values returned in the structure pointed to by *statusinfo* shall have been appropriately
 30900 read to ensure that the returned values are consistent. If the Trace Log option is supported and
 30901 the *trid* argument refers to a pre-recorded trace stream, the status shall be the status of the
 30902 completed trace stream.

30903 Each time the *posix_trace_get_status()* function is used, the overrun status of the trace stream
 30904 shall be reset to **POSIX_TRACE_NO_OVERRUN** immediately after the call completes. If the
 30905 Trace Log option is supported, the *posix_trace_get_status()* function shall behave the same as
 30906 when the option is not supported except for the following differences:

- If the *trid* argument refers to a trace stream with log, each time the *posix_trace_get_status()*
 30908 function is used, the log overrun status of the trace stream shall be reset to
 30909 **POSIX_TRACE_NO_OVERRUN** and the *flush_error* status shall be reset to zero immediately
 30910 after the call completes.
- If the *trid* argument refers to a pre-recorded trace stream, the status returned shall be the
 30912 status of the completed trace stream and the status values of the trace stream shall not be
 30913 reset.

30914

30915 RETURN VALUE

30916 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 30917 return the corresponding error number.

30918 The *posix_trace_get_attr()* function stores the trace attributes in the object pointed to by *attr*, if
 30919 successful.

30920 The *posix_trace_get_status()* function stores the trace status in the object pointed to by *statusinfo*,
 30921 if successful.

30922 ERRORS

30923 These functions shall fail if:

30924 [EINVAL] The trace stream argument *trid* does not correspond to a valid active trace
 30925 stream or a valid trace log.

30926 EXAMPLES

30927 None.

30928 APPLICATION USAGE

30929 None.

30930 RATIONALE

30931 None.

30932 FUTURE DIRECTIONS

30933 None.

30934 SEE ALSO

30935 *posix_trace_attr_destroy()*, *posix_trace_attr_init()*, *posix_trace_create()*, *posix_trace_open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <trace.h>

30937 CHANGE HISTORY

30938 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30939 IEEE PASC Interpretation 1003.1 #123 is applied.

30940 NAME

30941 posix_trace_get_filter, posix_trace_set_filter — retrieve and set the filter of an initialized trace
 30942 stream (**TRACING**)

30943 SYNOPSIS

```
30944 TRC TEF #include <trace.h>
30945     int posix_trace_get_filter(trace_id_t trid, trace_event_set_t *set);
30946     int posix_trace_set_filter(trace_id_t trid,
30947         const trace_event_set_t *set, int how);
```

30949 DESCRIPTION

30950 The *posix_trace_get_filter()* function shall retrieve, into the argument pointed to by *set*, the actual
 30951 trace event filter from the trace stream specified by *trid*.

30952 The *posix_trace_set_filter()* function shall change the set of filtered trace event types after a trace
 30953 stream identified by the *trid* argument is created. This function may be called prior to starting
 30954 the trace stream, or while the trace stream is active. By default, if no call is made to
 30955 *posix_trace_set_filter()*, all trace events shall be recorded (that is, none of the trace event types are
 30956 filtered out).

30957 If this function is called while the trace is in progress, a special system trace event,
 30958 **POSIX_TRACE_FILTER**, shall be recorded in the trace indicating both the old and the new sets
 30959 of filtered trace event types (see Table 2-4 (on page 79) and Table 2-6 (on page 80)).

30960 If the *posix_trace_set_filter()* function is interrupted by a signal, an error shall be returned and the
 30961 filter shall not be changed. In this case, the state of the trace stream shall not be changed.

30962 The value of the argument *how* indicates the manner in which the set is to be changed and shall
 30963 have one of the following values, as defined in the <trace.h> header:

30964 **POSIX_TRACE_SET_EVENTSET**

30965 The resulting set of trace event types to be filtered shall be the trace event type set pointed
 30966 to by the argument *set*.

30967 **POSIX_TRACE_ADD_EVENTSET**

30968 The resulting set of trace event types to be filtered shall be the union of the current set and
 30969 the trace event type set pointed to by the argument *set*.

30970 **POSIX_TRACE_SUB_EVENTSET**

30971 The resulting set of trace event types to be filtered shall be all trace event types in the
 30972 current set that are not in the set pointed to by the argument *set*; that is, remove each
 30973 element of the specified set from the current filter.

30974 RETURN VALUE

30975 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
 30976 return the corresponding error number.

30977 The *posix_trace_get_filter()* function stores the set of filtered trace event types in *set*, if successful.

30978 ERRORS

30979 These functions shall fail if:

30980	[EINVAL]	The value of the <i>trid</i> argument does not correspond to an active trace stream or the value of the argument pointed to by <i>set</i> is invalid.
30982	[EINTR]	The operation was interrupted by a signal.

30983 **EXAMPLES**

30984 None.

30985 **APPLICATION USAGE**

30986 None.

30987 **RATIONALE**

30988 None.

30989 **FUTURE DIRECTIONS**

30990 None.

30991 **SEE ALSO**

30992 Table 2-4 (on page 79), Table 2-6 (on page 80), *posix_trace_eventset_add()*, the Base Definitions
30993 volume of IEEE Std 1003.1-2001, <trace.h>

30994 **CHANGE HISTORY**

30995 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

30996 IEEE PASC Interpretation 1003.1 #123 is applied.

30997 NAME

30998 **posix_trace_get_status** — retrieve the trace status (**TRACING**)

30999 SYNOPSIS

31000 TRC `#include <trace.h>`

31001 `int posix_trace_get_status(trace_id_t trid,`
31002 `struct posix_trace_status_info *statusinfo);`

31003

31004 DESCRIPTION

31005 Refer to *posix_trace_get_attr()*.

31006 NAME

31007 posix_trace_getnext_event, posix_trace_timedgetnext_event, posix_trace_trygetnext_event —
 31008 retrieve a trace event (**TRACING**)

31009 SYNOPSIS

```
31010 TRC #include <sys/types.h>
31011 #include <trace.h>

31012 int posix_trace_getnext_event(trace_id_t trid,
31013         struct posix_trace_event_info *restrict event,
31014         void *restrict data, size_t num_bytes,
31015         size_t *restrict data_len, int *restrict unavailable);
31016 TMC int posix_trace_timedgetnext_event(trace_id_t trid,
31017         struct posix_trace_event_info *restrict event,
31018         void *restrict data, size_t num_bytes,
31019         size_t *restrict data_len, int *restrict unavailable,
31020         const struct timespec *restrict abs_timeout);
31021 TRC int posix_trace_trygetnext_event(trace_id_t trid,
31022         struct posix_trace_event_info *restrict event,
31023         void *restrict data, size_t num_bytes,
31024         size_t *restrict data_len, int *restrict unavailable);
31025
```

31026 DESCRIPTION

31027 The *posix_trace_getnext_event()* function shall report a recorded trace event either from an active
 31028 trace stream without log or a pre-recorded trace stream identified by the *trid* argument. The
 31029 *posix_trace_trygetnext_event()* function shall report a recorded trace event from an active trace
 31030 stream without log identified by the *trid* argument.

31031 The trace event information associated with the recorded trace event shall be copied by the
 31032 function into the structure pointed to by the argument *event* and the data associated with the
 31033 trace event shall be copied into the buffer pointed to by the *data* argument.

31034 The *posix_trace_getnext_event()* function shall block if the *trid* argument identifies an active trace
 31035 stream and there is currently no trace event ready to be retrieved. When returning, if a recorded
 31036 trace event was reported, the variable pointed to by the *unavailable* argument shall be set to zero.
 31037 Otherwise, the variable pointed to by the *unavailable* argument shall be set to a value different
 31038 from zero.

31039 TMO The *posix_trace_timedgetnext_event()* function shall attempt to get another trace event from an
 31040 active trace stream without log, as in the *posix_trace_getnext_event()* function. However, if no
 31041 trace event is available from the trace stream, the implied wait shall be terminated when the
 31042 timeout specified by the argument *abs_timeout* expires, and the function shall return the error
 31043 [*ETIMEDOUT*].

31044 The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by
 31045 the clock upon which timeouts are based (that is, when the value of that clock equals or exceeds
 31046 *abs_timeout*), or if the absolute time specified by *abs_timeout* has already passed at the time of the
 31047 call.

31048 TMO TMR If the Timers option is supported, the timeout shall be based on the *CLOCK_REALTIME* clock;
 31049 if the Timers option is not supported, the timeout shall be based on the system clock as returned
 31050 by the *time()* function. The resolution of the timeout shall be the resolution of the clock on which
 31051 it is based. The **timespec** data type is defined in the <time.h> header.

31052 TMO Under no circumstance shall the function fail with a timeout if a trace event is immediately
 31053 available from the trace stream. The validity of the *abs_timeout* argument need not be checked if

31054 a trace event is immediately available from the trace stream.

31055 The behavior of this function for a pre-recorded trace stream is unspecified.

31056 TRL The *posix_trace_trygetnext_event()* function shall not block. This function shall return an error if
31057 the *trid* argument identifies a pre-recorded trace stream. If a recorded trace event was reported,
31058 the variable pointed to by the *unavailable* argument shall be set to zero. Otherwise, if no trace
31059 event was reported, the variable pointed to by the *unavailable* argument shall be set to a value
31060 different from zero.

31061 The argument *num_bytes* shall be the size of the buffer pointed to by the *data* argument. The
31062 argument *data_len* reports to the application the length in bytes of the data record just
31063 transferred. If *num_bytes* is greater than or equal to the size of the data associated with the trace
31064 event pointed to by the *event* argument, all the recorded data shall be transferred. In this case, the
31065 *truncation-status* member of the trace event structure shall be either
31066 POSIX_TRACE_NOT_TRUNCATED, if the trace event data was recorded without truncation
31067 while tracing, or POSIX_TRACE_TRUNCATED_RECORD, if the trace event data was truncated
31068 when it was recorded. If the *num_bytes* argument is less than the length of recorded trace event
31069 data, the data transferred shall be truncated to a length of *num_bytes*, the value stored in the
31070 variable pointed to by *data_len* shall be equal to *num_bytes*, and the *truncation-status* member of
31071 the *event* structure argument shall be set to POSIX_TRACE_TRUNCATED_READ (see the
31072 *posix_trace_event_info* structure defined in <trace.h>).

31073 The report of a trace event shall be sequential starting from the oldest recorded trace event. Trace
31074 events shall be reported in the order in which they were generated, up to an implementation-
31075 defined time resolution that causes the ordering of trace events occurring very close to each
31076 other to be unknown. Once reported, a trace event cannot be reported again from an active trace
31077 stream. Once a trace event is reported from an active trace stream without log, the trace stream
31078 shall make the resources associated with that trace event available to record future generated
31079 trace events.

31080 **RETURN VALUE**

31081 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
31082 return the corresponding error number.

31083 If successful, these functions store:

31084 • The recorded trace event in the object pointed to by *event*

31085 • The trace event information associated with the recorded trace event in the object pointed to
31086 by *data*

31087 • The length of this trace event information in the object pointed to by *data_len*

31088 • The value of zero in the object pointed to by *unavailable*

31089 **ERRORS**

31090 These functions shall fail if:

31091 [EINVAL] The trace stream identifier argument *trid* is invalid.

31092 The *posix_trace_getnext_event()* and *posix_trace_timedgetnext_event()* functions shall fail if:

31093 [EINTR] The operation was interrupted by a signal, and so the call had no effect.

31094 The *posix_trace_trygetnext_event()* function shall fail if:

31095 [EINVAL] The trace stream identifier argument *trid* does not correspond to an active
31096 trace stream.

31097 TMO The *posix_trace_timedgetnext_event()* function shall fail if:

31098 [EINVAL] There is no trace event immediately available from the trace stream, and the
31099 *timeout* argument is invalid.

31100 [ETIMEDOUT] No trace event was available from the trace stream before the specified
31101 *timeout* *timeout* expired.
31102

31103 EXAMPLES

31104 None.

31105 APPLICATION USAGE

31106 None.

31107 RATIONALE

31108 None.

31109 FUTURE DIRECTIONS

31110 None.

31111 SEE ALSO

31112 *posix_trace_create()*, *posix_trace_open()*, the Base Definitions volume of IEEE Std 1003.1-2001,
31113 <sys/types.h>, <trace.h>

31114 CHANGE HISTORY

31115 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

31116 IEEE PASC Interpretation 1003.1 #123 is applied.

31117 NAME

31118 **posix_trace_open**, **posix_trace_rewind** — trace log management (**TRACING**)

31119 SYNOPSIS

31120 TRC TRL `#include <trace.h>`

1

31121 `int posix_trace_open(int file_desc, trace_id_t *trid);`

31122 `int posix_trace_rewind(trace_id_t trid);`

31123

31124 DESCRIPTION

31125 Refer to *posix_trace_close()*.

31126 **NAME**31127 **posix_trace_set_filter** — set filter of an initialized trace stream (**TRACING**)31128 **SYNOPSIS**

31129 TRC TEF #include <trace.h>

31130 int posix_trace_set_filter(trace_id_t *trid*,31131 const trace_event_set_t **set*, int *how*);

31132

31133 **DESCRIPTION**31134 Refer to *posix_trace_get_filter()*.

31135 **NAME**31136 posix_trace_shutdown — trace stream shutdown from a process (**TRACING**)31137 **SYNOPSIS**

```
31138 TRC #include <sys/types.h>
31139 #include <trace.h>
31140 int posix_trace_shutdown(trace_id_t trid);
31141
```

31142 **DESCRIPTION**31143 Refer to *posix_trace_create()*.

31144 NAME

31145 **posix_trace_start**, **posix_trace_stop** — trace start and stop (**TRACING**)

31146 SYNOPSIS

31147 TRC `#include <trace.h>`

31148 `int posix_trace_start(trace_id_t trid);`

31149 `int posix_trace_stop (trace_id_t trid);`

31150

31151 DESCRIPTION

31152 The *posix_trace_start()* and *posix_trace_stop()* functions, respectively, shall start and stop the
31153 trace stream identified by the argument *trid*.

31154 The effect of calling the *posix_trace_start()* function shall be recorded in the trace stream as the
31155 POSIX_TRACE_START system trace event and the status of the trace stream shall become
31156 POSIX_TRACE_RUNNING. If the trace stream is in progress when this function is called, the
31157 POSIX_TRACE_START system trace event shall not be recorded and the trace stream shall
31158 continue to run. If the trace stream is full, the POSIX_TRACE_START system trace event shall
31159 not be recorded and the status of the trace stream shall not be changed.

31160 The effect of calling the *posix_trace_stop()* function shall be recorded in the trace stream as the
31161 POSIX_TRACE_STOP system trace event and the status of the trace stream shall become
31162 POSIX_TRACE_SUSPENDED. If the trace stream is suspended when this function is called, the
31163 POSIX_TRACE_STOP system trace event shall not be recorded and the trace stream shall remain
31164 suspended. If the trace stream is full, the POSIX_TRACE_STOP system trace event shall not be
31165 recorded and the status of the trace stream shall not be changed.

31166 RETURN VALUE

31167 Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
31168 return the corresponding error number.

31169 ERRORS

31170 These functions shall fail if:

31171 [EINVAL] The value of the argument *trid* does not correspond to an active trace stream
31172 and thus no trace stream was started or stopped.

31173 [EINTR] The operation was interrupted by a signal and thus the trace stream was not
31174 necessarily started or stopped.

31175 EXAMPLES

31176 None.

31177 APPLICATION USAGE

31178 None.

31179 RATIONALE

31180 None.

31181 FUTURE DIRECTIONS

31182 None.

31183 SEE ALSO

31184 *posix_trace_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<trace.h>**

31185 CHANGE HISTORY

31186 First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

31187 IEEE PASC Interpretation 1003.1 #123 is applied.

31188 **NAME**31189 **posix_trace_timedgetnext_event**, — retrieve a trace event (**TRACING**)31190 **SYNOPSIS**

```
31191 TRC TMO #include <sys/types.h>
31192     #include <trace.h>
31193     int posix_trace_timedgetnext_event(trace_id_t trid,
31194         struct posix_trace_event_info *restrict event,
31195         void *restrict data, size_t num_bytes,
31196         size_t *restrict data_len, int *restrict unavailable,
31197         const struct timespec *restrict abs_timeout);
```

31198

31199 **DESCRIPTION**31200 Refer to *posix_trace_getnext_event()*.

31201 **NAME**31202 **posix_trace_trid_eventid_open** — open a trace event type identifier (**TRACING**)31203 **SYNOPSIS**

31204 TRC TEF #include <trace.h>

31205 int posix_trace_trid_eventid_open(trace_id_t *trid*,
31206 const char *restrict *event_name*,
31207 trace_event_id_t *restrict *event*);
3120831209 **DESCRIPTION**31210 Refer to *posix_trace_eventid_equal()*.

31211 NAME

31212 **posix_trace_trygetnext_event** — retrieve a trace event (**TRACING**)

31213 SYNOPSIS

```
31214 TRC #include <sys/types.h>
31215 #include <trace.h>
31216 int posix_trace_trygetnext_event(trace_id_t trid,
31217     struct posix_trace_event_info *restrict event,
31218     void *restrict data, size_t num_bytes,
31219     size_t *restrict data_len, int *restrict unavailable);
31220
```

31221 DESCRIPTION

31222 Refer to *posix_trace_getnext_event()*.

31223 NAME

31224 posix_typed_mem_get_info — query typed memory information (ADVANCED REALTIME)

31225 SYNOPSIS

31226 TYM #include <sys/mman.h>

```
31227      int posix_typed_mem_get_info(int fildes,  
31228            struct posix_typed_mem_info *info);
```

31229

31230 DESCRIPTION

31231 The *posix_typed_mem_get_info()* function shall return, in the *posix_tmi_length* field of the
31232 **posix_typed_mem_info** structure pointed to by *info*, the maximum length which may be
31233 successfully allocated by the typed memory object designated by *fildes*. This maximum length
31234 shall take into account the flag **POSIX_TYPED_MEM_ALLOCATE** or
31235 **POSIX_TYPED_MEM_ALLOCATE_CONTIG** specified when the typed memory object
31236 represented by *fildes* was opened. The maximum length is dynamic; therefore, the value returned
31237 is valid only while the current mapping of the corresponding typed memory pool remains
31238 unchanged.

31239 If *fildes* represents a typed memory object opened with neither the
31240 **POSIX_TYPED_MEM_ALLOCATE** flag nor the **POSIX_TYPED_MEM_ALLOCATE_CONTIG**
31241 flag specified, the returned value of *info->posix_tmi_length* is unspecified.

31242 The *posix_typed_mem_get_info()* function may return additional implementation-defined
31243 information in other fields of the **posix_typed_mem_info** structure pointed to by *info*.

31244 If the memory object specified by *fildes* is not a typed memory object, then the behavior of this
31245 function is undefined.

31246 RETURN VALUE

31247 Upon successful completion, the *posix_typed_mem_get_info()* function shall return zero;
31248 otherwise, the corresponding error status value shall be returned.

31249 ERRORS

31250 The *posix_typed_mem_get_info()* function shall fail if:

31251 [EBADF] The *fildes* argument is not a valid open file descriptor.

31252 [ENODEV] The *fildes* argument is not connected to a memory object supported by this
31253 function.

31254 This function shall not return an error code of [EINTR].

31255 EXAMPLES

31256 None.

31257 APPLICATION USAGE

31258 None.

31259 RATIONALE

31260 An application that needs to allocate a block of typed memory with length dependent upon the
31261 amount of memory currently available must either query the typed memory object to obtain the
31262 amount available, or repeatedly invoke *mmap()* attempting to guess an appropriate length.
31263 While the latter method is existing practice with *malloc()*, it is awkward and imprecise. The
31264 *posix_typed_mem_get_info()* function allows an application to immediately determine available
31265 memory. This is particularly important for typed memory objects that may in some cases be
31266 scarce resources. Note that when a typed memory pool is a shared resource, some form of
31267 mutual-exclusion or synchronization may be required while typed memory is being queried and

31268 allocated to prevent race conditions.

31269 The existing *fstat()* function is not suitable for this purpose. We realize that implementations
31270 may wish to provide other attributes of typed memory objects (for example, alignment
31271 requirements, page size, and so on). The *fstat()* function returns a structure which is not
31272 extensible and, furthermore, contains substantial information that is inappropriate for typed
31273 memory objects.

31274 **FUTURE DIRECTIONS**

31275 None.

31276 **SEE ALSO**

31277 *fstat()*, *mmap()*, *posix_typed_mem_open()*, the Base Definitions volume of IEEE Std 1003.1-2001,
31278 <sys/mman.h>

31279 **CHANGE HISTORY**

31280 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

31281 NAME

31282 posix_typed_mem_open — open a typed memory object (**ADVANCED REALTIME**)

31283 SYNOPSIS

31284 TTY #include <sys/mman.h>

31285 int posix_typed_mem_open(const char *name, int oflag, int tflag);

31286

31287 DESCRIPTION

The *posix_typed_mem_open()* function shall establish a connection between the typed memory object specified by the string pointed to by *name* and a file descriptor. It shall create an open file description that refers to the typed memory object and a file descriptor that refers to that open file description. The file descriptor is used by other functions to refer to that typed memory object. It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments. The *name* argument shall conform to the construction rules for a pathname. If *name* begins with the slash character, then processes calling *posix_typed_mem_open()* with the same value of *name* shall refer to the same typed memory object. If *name* does not begin with the slash character, the effect is implementation-defined. The interpretation of slash characters other than the leading slash character in *name* is implementation-defined.

Each typed memory object supported in a system shall be identified by a name which specifies not only its associated typed memory pool, but also the path or port by which it is accessed. That is, the same typed memory pool accessed via several different ports shall have several different corresponding names. The binding between names and typed memory objects is established in an implementation-defined manner. Unlike shared memory objects, there is no way within IEEE Std 1003.1-2001 for a program to create a typed memory object.

The value of *tflag* shall determine how the typed memory object behaves when subsequently mapped by calls to *mmap()*. At most, one of the following flags defined in <sys/mman.h> may be specified:

31308 POSIX_TYPED_MEM_ALLOCATE
31309 Allocate on *mmap()*.

31310 POSIX_TYPED_MEM_ALLOCATE_CONTIG
31311 Allocate contiguously on *mmap()*.

31312 POSIX_TYPED_MEM_MAP_ALLOCATABLE
31313 Map on *mmap()*, without affecting allocatability.

If *tflag* has the flag *POSIX_TYPED_MEM_ALLOCATE* specified, any subsequent call to *mmap()* using the returned file descriptor shall result in allocation and mapping of typed memory from the specified typed memory pool. The allocated memory may be a contiguous previously unallocated area of the typed memory pool or several non-contiguous previously unallocated areas (mapped to a contiguous portion of the process address space). If *tflag* has the flag *POSIX_TYPED_MEM_ALLOCATE_CONTIG* specified, any subsequent call to *mmap()* using the returned file descriptor shall result in allocation and mapping of a single contiguous previously unallocated area of the typed memory pool (also mapped to a contiguous portion of the process address space). If *tflag* has none of the flags *POSIX_TYPED_MEM_ALLOCATE* or *POSIX_TYPED_MEM_ALLOCATE_CONTIG* specified, any subsequent call to *mmap()* using the returned file descriptor shall map an application-chosen area from the specified typed memory pool such that this mapped area becomes unavailable for allocation until unmapped by all processes. If *tflag* has the flag *POSIX_TYPED_MEM_MAP_ALLOCATABLE* specified, any subsequent call to *mmap()* using the returned file descriptor shall map an application-chosen area from the specified typed memory pool without an effect on the availability of that area for

31329 allocation; that is, mapping such an object leaves each byte of the mapped area unallocated if it
31330 was unallocated prior to the mapping or allocated if it was allocated prior to the mapping. The
31331 appropriate privilege to specify the `POSIX_TYPED_MEM_MAP_ALLOCATABLE` flag is
31332 implementation-defined.

31333 If successful, `posix_typed_mem_open()` shall return a file descriptor for the typed memory object
31334 that is the lowest numbered file descriptor not currently open for that process. The open file
31335 description is new, and therefore the file descriptor shall not share it with any other processes. It
31336 is unspecified whether the file offset is set. The `FD_CLOEXEC` file descriptor flag associated
31337 with the new file descriptor shall be cleared.

31338 The behavior of `msync()`, `ftruncate()`, and all file operations other than `mmap()`,
31339 `posix_mem_offset()`, `posix_typed_mem_get_info()`, `fstat()`, `dup()`, `dup2()`, and `close()`, is unspecified
31340 when passed a file descriptor connected to a typed memory object by this function.

31341 The file status flags of the open file description shall be set according to the value of `oflag`.
31342 Applications shall specify exactly one of the three access mode values described below and
31343 defined in the `<fcntl.h>` header, as the value of `oflag`.

31344 `O_RDONLY` Open for read access only.

31345 `O_WRONLY` Open for write access only.

31346 `O_RDWR` Open for read or write access.

31347 RETURN VALUE

31348 Upon successful completion, the `posix_typed_mem_open()` function shall return a non-negative
31349 integer representing the lowest numbered unused file descriptor. Otherwise, it shall return `-1`
31350 and set `errno` to indicate the error.

31351 ERRORS

31352 The `posix_typed_mem_open()` function shall fail if:

31353 `[EACCES]` The typed memory object exists and the permissions specified by `oflag` are
31354 denied.

31355 `[EINTR]` The `posix_typed_mem_open()` operation was interrupted by a signal.

31356 `[EINVAL]` The flags specified in `tflag` are invalid (more than one of
31357 `POSIX_TYPED_MEM_ALLOCATE`,
31358 `POSIX_TYPED_MEM_ALLOCATE_CONTIG`, or
31359 `POSIX_TYPED_MEM_MAP_ALLOCATABLE` is specified).

31360 `[EMFILE]` Too many file descriptors are currently in use by this process.

31361 `[ENAMETOOLONG]` The length of the `name` argument exceeds `{PATH_MAX}` or a pathname
31362 component is longer than `{NAME_MAX}`.

31364 `[ENFILE]` Too many file descriptors are currently open in the system.

31365 `[ENOENT]` The named typed memory object does not exist.

31366 `[EPERM]` The caller lacks the appropriate privilege to specify the flag
31367 `POSIX_TYPED_MEM_MAP_ALLOCATABLE` in argument `tflag`.

31368 EXAMPLES

31369 None.

31370 APPLICATION USAGE

31371 None.

31372 RATIONALE

31373 None.

31374 FUTURE DIRECTIONS

31375 None.

31376 SEE ALSO

31377 *close()*, *dup()*, *exec*, *fcntl()*, *fstat()*, *ftruncate()*, *mmap()*, *msync()*, *posix_mem_offset()*,
31378 *posix_typed_mem_get_info()*, *umask()*, the Base Definitions volume of IEEE Std 1003.1-2001,
31379 <*fcntl.h*>, <*sys/mman.h*>

31380 CHANGE HISTORY

31381 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

31382 NAME

31383 pow, powf, powl — power function

31384 SYNOPSIS

```
31385     #include <math.h>
31386
31387     double pow(double x, double y);
31388     float powf(float x, float y);
31389     long double powl(long double x, long double y);
```

31389 DESCRIPTION

31390 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 31391 conflict between the requirements described here and the ISO C standard is unintentional. This
 31392 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

31393 These functions shall compute the value of x raised to the power y , x^y . If x is negative, the
 31394 application shall ensure that y is an integer value.

31395 An application wishing to check for error situations should set *errno* to zero and call
 31396 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
 31397 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
 31398 zero, an error has occurred.

31399 RETURN VALUE

31400 Upon successful completion, these functions shall return the value of x raised to the power y .

31401 MX For finite values of $x < 0$, and finite non-integer values of y , a domain error shall occur and either
 31402 a NaN (if representable), or an implementation-defined value shall be returned.

31403 If the correct value would cause overflow, a range error shall occur and *pow()*, *powf()*, and
 31404 *powl()* shall return $\pm\text{HUGE_VAL}$, $\pm\text{HUGE_VALF}$, and $\pm\text{HUGE_VALL}$, respectively, with the
 31405 same sign as the correct value of the function. 1

31406 If the correct value would cause underflow, and is not representable, a range error may occur,
 31407 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

31408 MX If x or y is a NaN, a NaN shall be returned (unless specified elsewhere in this description).

31409 For any value of y (including NaN), if x is +1, 1.0 shall be returned.

31410 For any value of x (including NaN), if y is ± 0 , 1.0 shall be returned.

31411 For any odd integer value of $y > 0$, if x is ± 0 , ± 0 shall be returned.

31412 For $y > 0$ and not an odd integer, if x is ± 0 , $+0$ shall be returned.

31413 If x is -1 , and y is $\pm\text{Inf}$, 1.0 shall be returned.

31414 For $|x| < 1$, if y is $-\text{Inf}$, $+\text{Inf}$ shall be returned.

31415 For $|x| > 1$, if y is $-\text{Inf}$, $+0$ shall be returned.

31416 For $|x| < 1$, if y is $+\text{Inf}$, $+0$ shall be returned.

31417 For $|x| > 1$, if y is $+\text{Inf}$, $+\text{Inf}$ shall be returned.

31418 For y an odd integer < 0 , if x is $-\text{Inf}$, -0 shall be returned.

31419 For $y < 0$ and not an odd integer, if x is $-\text{Inf}$, $+0$ shall be returned.

31420 For y an odd integer > 0 , if x is $-\text{Inf}$, $-\text{Inf}$ shall be returned.

31421 For $y > 0$ and not an odd integer, if x is $-\text{Inf}$, $+\text{Inf}$ shall be returned.

31422 For $y < 0$, if x is $+\infty$, $+0$ shall be returned.

31423 For $y > 0$, if x is $+\infty$, $+\infty$ shall be returned.

31424 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 $\pm\text{HUGE_VALL}$ shall be returned for `pow()`, `powf()`, and `powl()`, respectively.

31425 For $y < 0$ and not an odd integer, if x is ± 0 , a pole error shall occur and HUGE_VAL , HUGE_VALF , and HUGE_VALL shall be returned for `pow()`, `powf()`, and `powl()`, respectively.

31426 If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

31427

31428

31429

31430 ERRORS

31431 These functions shall fail if:

31432 Domain Error The value of x is negative and y is a finite non-integer.
31433 If the integer expression (`math_errhandling & MATH_ERRNO`) is non-zero,
31434 then `errno` shall be set to [EDOM]. If the integer expression (`math_errhandling & MATH_ERREXCEPT`) is non-zero, then the invalid floating-point exception
31435 shall be raised.

31436

31437 MX Pole Error The value of x is zero and y is negative.
31438 If the integer expression (`math_errhandling & MATH_ERRNO`) is non-zero,
31439 then `errno` shall be set to [ERANGE]. If the integer expression
31440 (`math_errhandling & MATH_ERREXCEPT`) is non-zero, then the divide-by-
31441 zero floating-point exception shall be raised.

31442 Range Error The result overflows.
31443 If the integer expression (`math_errhandling & MATH_ERRNO`) is non-zero,
31444 then `errno` shall be set to [ERANGE]. If the integer expression
31445 (`math_errhandling & MATH_ERREXCEPT`) is non-zero, then the overflow
31446 floating-point exception shall be raised.

31447 These functions may fail if:

31448 Range Error The result underflows.
31449 If the integer expression (`math_errhandling & MATH_ERRNO`) is non-zero,
31450 then `errno` shall be set to [ERANGE]. If the integer expression
31451 (`math_errhandling & MATH_ERREXCEPT`) is non-zero, then the underflow
31452 floating-point exception shall be raised.

31453 EXAMPLES

31454 None.

31455 APPLICATION USAGE

31456 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling & MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

31457

31458 RATIONALE

31459 None.

31460 FUTURE DIRECTIONS

31461 None.

31462 SEE ALSO

31463 *exp()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
31464 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

31465 CHANGE HISTORY

31466 First released in Issue 1. Derived from Issue 1 of the SVID.

31467 Issue 5

31468 The DESCRIPTION is updated to indicate how an application should check for an error. This
31469 text was previously published in the APPLICATION USAGE section.

31470 Issue 6

31471 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

31472 The *powf()* and *powl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

31473 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
31474 revised to align with the ISO/IEC 9899:1999 standard.

31475 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
31476 marked.

31477 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/42 is applied, correcting the third 1
31478 paragraph in the RETURN VALUE section. 1

31479 **NAME**31480 **pread** — read from a file31481 **SYNOPSIS**

31482 XSI #include <unistd.h>

31483 ssize_t pread(int *fd*, void **buf*, size_t *nbyte*, off_t *offset*);

31484

31485 **DESCRIPTION**31486 Refer to *read()*.

31487 **NAME**
31488 **printf** — print formatted output

31489 **SYNOPSIS**
31490 #include <stdio.h>
31491 int printf(const char *restrict *format*, ...);

31492 **DESCRIPTION**
31493 Refer to *fprintf()*.

31494 NAME

31495 pselect, select — synchronous I/O multiplexing

31496 SYNOPSIS

```
31497 #include <sys/select.h>
31498 int pselect(int nfds, fd_set *restrict readfds,
31499     fd_set *restrict writefds, fd_set *restrict errorfds,
31500     const struct timespec *restrict timeout,
31501     const sigset_t *restrict sigmask);
31502 int select(int nfds, fd_set *restrict readfds,
31503     fd_set *restrict writefds, fd_set *restrict errorfds,
31504     struct timeval *restrict timeout);
31505 void FD_CLR(int fd, fd_set *fdset);
31506 int FD_ISSET(int fd, fd_set *fdset);
31507 void FD_SET(int fd, fd_set *fdset);
31508 void FD_ZERO(fd_set *fdset);
```

31509 DESCRIPTION

31510 The *pselect()* function shall examine the file descriptor sets whose addresses are passed in the
 31511 *readfds*, *writefds*, and *errorfds* parameters to see whether some of their descriptors are ready for
 31512 reading, are ready for writing, or have an exceptional condition pending, respectively.

31513 The *select()* function shall be equivalent to the *pselect()* function, except as follows:

- 31514 • For the *select()* function, the timeout period is given in seconds and microseconds in an
 31515 argument of type **struct timeval**, whereas for the *pselect()* function the timeout period is
 31516 given in seconds and nanoseconds in an argument of type **struct timespec**.
- 31517 • The *select()* function has no *sigmask* argument; it shall behave as *pselect()* does when *sigmask*
 31518 is a null pointer.
- 31519 • Upon successful completion, the *select()* function may modify the object pointed to by the
 31520 *timeout* argument.

31521 The *pselect()* and *select()* functions shall support regular files, terminal and pseudo-terminal
 31522 devices, STREAMS-based files, FIFOs, pipes, and sockets. The behavior of *pselect()* and *select()*
 31523 on file descriptors that refer to other types of file is unspecified.

31524 The *nfds* argument specifies the range of descriptors to be tested. The first *nfds* descriptors shall
 31525 be checked in each set; that is, the descriptors from zero through *nfds*-1 in the descriptor sets
 31526 shall be examined.

31527 If the *readfds* argument is not a null pointer, it points to an object of type **fd_set** that on input
 31528 specifies the file descriptors to be checked for being ready to read, and on output indicates
 31529 which file descriptors are ready to read.

31530 If the *writefds* argument is not a null pointer, it points to an object of type **fd_set** that on input
 31531 specifies the file descriptors to be checked for being ready to write, and on output indicates
 31532 which file descriptors are ready to write.

31533 If the *errorfds* argument is not a null pointer, it points to an object of type **fd_set** that on input
 31534 specifies the file descriptors to be checked for error conditions pending, and on output indicates
 31535 which file descriptors have error conditions pending.

31536 Upon successful completion, the *pselect()* or *select()* function shall modify the objects pointed to
 31537 by the *readfds*, *writefds*, and *errorfds* arguments to indicate which file descriptors are ready for
 31538 reading, ready for writing, or have an error condition pending, respectively, and shall return the
 31539 total number of ready descriptors in all the output sets. For each file descriptor less than *nfds*, the

31540 corresponding bit shall be set on successful completion if it was set on input and the associated
31541 condition is true for that file descriptor.

31542 If none of the selected descriptors are ready for the requested operation, the *pselect()* or *select()*
31543 function shall block until at least one of the requested operations becomes ready, until the
31544 *timeout* occurs, or until interrupted by a signal. The *timeout* parameter controls how long the
31545 *pselect()* or *select()* function shall take before timing out. If the *timeout* parameter is not a null
31546 pointer, it specifies a maximum interval to wait for the selection to complete. If the specified
31547 time interval expires without any requested operation becoming ready, the function shall return.
31548 If the *timeout* parameter is a null pointer, then the call to *pselect()* or *select()* shall block
31549 indefinitely until at least one descriptor meets the specified criteria. To effect a poll, the *timeout*
31550 parameter should not be a null pointer, and should point to a zero-valued **timespec** structure.

31551 The use of a timeout does not affect any pending timers set up by *alarm()*, *ualarm()*, or
31552 *setitimer()*.

31553 Implementations may place limitations on the maximum timeout interval supported. All
31554 implementations shall support a maximum timeout interval of at least 31 days. If the *timeout*
31555 argument specifies a timeout interval greater than the implementation-defined maximum value,
31556 the maximum value shall be used as the actual timeout value. Implementations may also place
31557 limitations on the granularity of timeout intervals. If the requested timeout interval requires a
31558 finer granularity than the implementation supports, the actual timeout interval shall be rounded
31559 up to the next supported value.

31560 If *sigmask* is not a null pointer, then the *pselect()* function shall replace the signal mask of the
31561 caller by the set of signals pointed to by *sigmask* before examining the descriptors, and shall
31562 restore the signal mask of the calling thread before returning. 2 2 2

31563 A descriptor shall be considered ready for reading when a call to an input function with
31564 O_NONBLOCK clear would not block, whether or not the function would transfer data
31565 successfully. (The function might return data, an end-of-file indication, or an error other than
31566 one indicating that it is blocked, and in each of these cases the descriptor shall be considered
31567 ready for reading.)

31568 A descriptor shall be considered ready for writing when a call to an output function with
31569 O_NONBLOCK clear would not block, whether or not the function would transfer data
31570 successfully.

31571 If a socket has a pending error, it shall be considered to have an exceptional condition pending.
31572 Otherwise, what constitutes an exceptional condition is file type-specific. For a file descriptor for
31573 use with a socket, it is protocol-specific except as noted below. For other file types it is
31574 implementation-defined. If the operation is meaningless for a particular file type, *pselect()* or
31575 *select()* shall indicate that the descriptor is ready for read or write operations, and shall indicate
31576 that the descriptor has no exceptional condition pending.

31577 If a descriptor refers to a socket, the implied input function is the *recvmsg()* function with
31578 parameters requesting normal and ancillary data, such that the presence of either type shall
31579 cause the socket to be marked as readable. The presence of out-of-band data shall be checked if
31580 the socket option SO_OOBINLINE has been enabled, as out-of-band data is enqueued with
31581 normal data. If the socket is currently listening, then it shall be marked as readable if an
31582 incoming connection request has been received, and a call to the *accept()* function shall complete
31583 without blocking.

31584 If a descriptor refers to a socket, the implied output function is the *sendmsg()* function supplying
31585 an amount of normal data equal to the current value of the SO_SNDLOWAT option for the
31586 socket. If a non-blocking call to the *connect()* function has been made for a socket, and the
31587 connection attempt has either succeeded or failed leaving a pending error, the socket shall be

31588 marked as writable.

31589 A socket shall be considered to have an exceptional condition pending if a receive operation with O_NONBLOCK clear for the open file description and with the MSG_OOB flag set would return out-of-band data without blocking. (It is protocol-specific whether the MSG_OOB flag would be used to read out-of-band data.) A socket shall also be considered to have an exceptional condition pending if an out-of-band data mark is present in the receive queue. Other circumstances under which a socket may be considered to have an exceptional condition pending are protocol-specific and implementation-defined.

31596 If the *readfds*, *writelfds*, and *errorfds* arguments are all null pointers and the *timeout* argument is not a null pointer, the *pselect()* or *select()* function shall block for the time specified, or until interrupted by a signal. If the *readfds*, *writelfds*, and *errorfds* arguments are all null pointers and the *timeout* argument is a null pointer, the *pselect()* or *select()* function shall block until interrupted by a signal.

31601 File descriptors associated with regular files shall always select true for ready to read, ready to write, and error conditions.

31603 On failure, the objects pointed to by the *readfds*, *writelfds*, and *errorfds* arguments shall not be modified. If the timeout interval expires without the specified condition being true for any of the specified file descriptors, the objects pointed to by the *readfds*, *writelfds*, and *errorfds* arguments shall have all bits set to 0.

31607 File descriptor masks of type **fd_set** can be initialized and tested with *FD_CLR()*, *FD_ISSET()*, *FD_SET()*, and *FD_ZERO()*. It is unspecified whether each of these is a macro or a function. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with any of these names, the behavior is undefined.

31611 *FD_CLR(fd, fdsetp)* shall remove the file descriptor *fd* from the set pointed to by *fdsetp*. If *fd* is not a member of this set, there shall be no effect on the set, nor will an error be returned.

31613 *FD_ISSET(fd, fdsetp)* shall evaluate to non-zero if the file descriptor *fd* is a member of the set pointed to by *fdsetp*, and shall evaluate to zero otherwise.

31615 *FD_SET(fd, fdsetp)* shall add the file descriptor *fd* to the set pointed to by *fdsetp*. If the file descriptor *fd* is already in this set, there shall be no effect on the set, nor will an error be returned.

31617 *FD_ZERO(fdsetp)* shall initialize the descriptor set pointed to by *fdsetp* to the null set. No error is returned if the set is not empty at the time *FD_ZERO()* is invoked.

31619 The behavior of these macros is undefined if the *fd* argument is less than 0 or greater than or equal to *FD_SETSIZE*, or if *fd* is not a valid file descriptor, or if any of the arguments are expressions with side effects.

31622 **RETURN VALUE**

31623 Upon successful completion, the *pselect()* and *select()* functions shall return the total number of bits set in the bit masks. Otherwise, -1 shall be returned, and *errno* shall be set to indicate the error.

31626 *FD_CLR()*, *FD_SET()*, and *FD_ZERO()* do not return a value. *FD_ISSET()* shall return a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.

31629 **ERRORS**

31630 Under the following conditions, *pselect()* and *select()* shall fail and set *errno* to:

31631 [EBADF] One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor.

31633	[EINTR]	The function was interrupted before any of the selected events occurred and before the timeout interval expired.
31634		
31635 XSI		If SA_RESTART has been set for the interrupting signal, it is implementation-defined whether the function restarts or returns with [EINTR].
31636		
31637	[EINVAL]	An invalid timeout interval was specified.
31638	[EINVAL]	The <i>nfds</i> argument is less than 0 or greater than FD_SETSIZE.
31639 XSR	[EINVAL]	One of the specified file descriptors refers to a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.
31640		

31641 EXAMPLES

31642 None.

31643 APPLICATION USAGE

31644 None.

31645 RATIONALE

31646 In previous versions of the Single UNIX Specification, the *select()* function was defined in the `<sys/time.h>` header. This is now changed to `<sys/select.h>`. The rationale for this change was
31647 as follows: the introduction of the *pselect()* function included the `<sys/select.h>` header and the
31648 `<sys/select.h>` header defines all the related definitions for the *pselect()* and *select()* functions.
31649 Backwards-compatibility to existing XSI implementations is handled by allowing `<sys/time.h>`
31650 to include `<sys/select.h>`.

31652 FUTURE DIRECTIONS

31653 None.

31654 SEE ALSO

31655 *accept()*, *alarm()*, *connect()*, *fcntl()*, *poll()*, *read()*, *recvmsg()*, *sendmsg()*, *setitimer()*, *ualarm()*,
31656 *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/select.h>`, `<sys/time.h>`

31657 CHANGE HISTORY

31658 First released in Issue 4, Version 2.

31659 Issue 5

31660 Moved from X/OPEN UNIX extension to BASE.

31661 In the ERRORS section, the text has been changed to indicate that [EINVAL] is returned when
31662 *nfds* is less than 0 or greater than FD_SETSIZE. It previously stated less than 0, or greater than or
31663 equal to FD_SETSIZE.

31664 Text about *timeout* is moved from the APPLICATION USAGE section to the DESCRIPTION.

31665 Issue 6

31666 The Open Group Corrigendum U026/6 is applied, changing the occurrences of *readfds* and *writefds*
31667 in the *select()* DESCRIPTION to be *readfds* and *writefds*.

31668 Text referring to sockets is added to the DESCRIPTION.

31669 The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are
31670 marked as part of the XSI STREAMS Option Group.

31671 The following new requirements on POSIX implementations derive from alignment with the
31672 Single UNIX Specification:

- These functions are now mandatory.

31674 The *pselect()* function is added for alignment with IEEE Std 1003.1g-2000 and additional detail
31675 related to sockets semantics is added to the DESCRIPTION.

31676	The <i>select()</i> function now requires inclusion of <sys/select.h>.	
31677	The restrict keyword is added to the <i>select()</i> prototype for alignment with the ISO/IEC 9899:1999 standard.	
31679	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/70 is applied, updating the DESCRIPTION to reference the signal mask in terms of the calling thread rather than the process.	2
31680		2
31681		2

31682 NAME

31683 pthread_atfork — register fork handlers

31684 SYNOPSIS

31685 THR #include <pthread.h>

```
31686 int pthread_atfork(void (*prepare)(void), void (*parent)(void),
31687     void (*child)(void));
```

31688

31689 DESCRIPTION

31690 The *pthread_atfork()* function shall declare fork handlers to be called before and after *fork()*, in
31691 the context of the thread that called *fork()*. The *prepare* fork handler shall be called before *fork()*
31692 processing commences. The *parent* fork handle shall be called after *fork()* processing completes
31693 in the parent process. The *child* fork handler shall be called after *fork()* processing completes in
31694 the child process. If no handling is desired at one or more of these three points, the
31695 corresponding fork handler address(es) may be set to NULL.

31696 The order of calls to *pthread_atfork()* is significant. The *parent* and *child* fork handlers shall be
31697 called in the order in which they were established by calls to *pthread_atfork()*. The *prepare* fork
31698 handlers shall be called in the opposite order.

31699 RETURN VALUE

31700 Upon successful completion, *pthread_atfork()* shall return a value of zero; otherwise, an error
31701 number shall be returned to indicate the error.

31702 ERRORS

31703 The *pthread_atfork()* function shall fail if:

31704 [ENOMEM] Insufficient table space exists to record the fork handler addresses.

31705 The *pthread_atfork()* function shall not return an error code of [EINTR].

31706 EXAMPLES

31707 None.

31708 APPLICATION USAGE

31709 None.

31710 RATIONALE

31711 There are at least two serious problems with the semantics of *fork()* in a multi-threaded
31712 program. One problem has to do with state (for example, memory) covered by mutexes.
31713 Consider the case where one thread has a mutex locked and the state covered by that mutex is
31714 inconsistent while another thread calls *fork()*. In the child, the mutex is in the locked state
31715 (locked by a nonexistent thread and thus can never be unlocked). Having the child simply
31716 reinitialize the mutex is unsatisfactory since this approach does not resolve the question about
31717 how to correct or otherwise deal with the inconsistent state in the child.

31718 It is suggested that programs that use *fork()* call an *exec* function very soon afterwards in the
31719 child process, thus resetting all states. In the meantime, only a short list of async-signal-safe
31720 library routines are promised to be available.

31721 Unfortunately, this solution does not address the needs of multi-threaded libraries. Application
31722 programs may not be aware that a multi-threaded library is in use, and they feel free to call any
31723 number of library routines between the *fork()* and *exec* calls, just as they always have. Indeed,
31724 they may be extant single-threaded programs and cannot, therefore, be expected to obey new
31725 restrictions imposed by the threads library.

31726 On the other hand, the multi-threaded library needs a way to protect its internal state during
31727 *fork()* in case it is re-entered later in the child process. The problem arises especially in multi-
31728 threaded I/O libraries, which are almost sure to be invoked between the *fork()* and *exec* calls to
31729 effect I/O redirection. The solution may require locking mutex variables during *fork()*, or it may
31730 entail simply resetting the state in the child after the *fork()* processing completes.

31731 The *pthread_atfork()* function provides multi-threaded libraries with a means to protect
31732 themselves from innocent application programs that call *fork()*, and it provides multi-threaded
31733 application programs with a standard mechanism for protecting themselves from *fork()* calls in
31734 a library routine or the application itself.

31735 The expected usage is that the *prepare* handler acquires all mutex locks and the other two fork
31736 handlers release them.

31737 For example, an application can supply a *prepare* routine that acquires the necessary mutexes the
31738 library maintains and supply *child* and *parent* routines that release those mutexes, thus ensuring
31739 that the child gets a consistent snapshot of the state of the library (and that no mutexes are left
31740 stranded). Alternatively, some libraries might be able to supply just a *child* routine that
31741 reinitializes the mutexes in the library and all associated states to some known value (for
31742 example, what it was when the image was originally executed).

31743 When *fork()* is called, only the calling thread is duplicated in the child process. Synchronization
31744 variables remain in the same state in the child as they were in the parent at the time *fork()* was
31745 called. Thus, for example, mutex locks may be held by threads that no longer exist in the child
31746 process, and any associated states may be inconsistent. The parent process may avoid this by
31747 explicit code that acquires and releases locks critical to the child via *pthread_atfork()*. In addition,
31748 any critical threads need to be recreated and reinitialized to the proper state in the child (also via
31749 *pthread_atfork()*).

31750 A higher-level package may acquire locks on its own data structures before invoking lower-level
31751 packages. Under this scenario, the order specified for fork handler calls allows a simple rule of
31752 initialization for avoiding package deadlock: a package initializes all packages on which it
31753 depends before it calls the *pthread_atfork()* function for itself.

31754 FUTURE DIRECTIONS

31755 None.

31756 SEE ALSO

31757 *atexit()*, *fork()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>

31758 CHANGE HISTORY

31759 First released in Issue 5. Derived from the POSIX Threads Extension.

31760 IEEE PASC Interpretation 1003.1c #4 is applied.

31761 Issue 6

31762 The *pthread_atfork()* function is marked as part of the Threads option.

31763 The <pthread.h> header is added to the SYNOPSIS.

31764 NAME

31765 *pthread_attr_destroy*, *pthread_attr_init* — destroy and initialize the thread attributes object

31766 SYNOPSIS

31767 THR *#include <pthread.h>*

31768 *int pthread_attr_destroy(pthread_attr_t *attr);*

31769 *int pthread_attr_init(pthread_attr_t *attr);*

31770

31771 DESCRIPTION

31772 The *pthread_attr_destroy()* function shall destroy a thread attributes object. An implementation
31773 may cause *pthread_attr_destroy()* to set *attr* to an implementation-defined invalid value. A
31774 destroyed *attr* attributes object can be reinitialized using *pthread_attr_init()*; the results of
31775 otherwise referencing the object after it has been destroyed are undefined.

31776 The *pthread_attr_init()* function shall initialize a thread attributes object *attr* with the default
31777 value for all of the individual attributes used by a given implementation.

31778 The resulting attributes object (possibly modified by setting individual attribute values) when
31779 used by *pthread_create()* defines the attributes of the thread created. A single attributes object can
31780 be used in multiple simultaneous calls to *pthread_create()*. Results are undefined if
31781 *pthread_attr_init()* is called specifying an already initialized *attr* attributes object.

31782 RETURN VALUE

31783 Upon successful completion, *pthread_attr_destroy()* and *pthread_attr_init()* shall return a value of
31784 0; otherwise, an error number shall be returned to indicate the error.

31785 ERRORS

31786 The *pthread_attr_init()* function shall fail if:

31787 [ENOMEM] Insufficient memory exists to initialize the thread attributes object.

31788 The *pthread_attr_destroy()* function may fail if:

2

31789 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
31790 object.

2

31791 The *pthread_attr_init()* function may fail if:

2

31792 [EBUSY] The implementation has detected an attempt to reinitialize the thread
31793 attribute referenced by *attr*, a previously initialized, but not yet destroyed,
31794 thread attribute.

2

31795 These functions shall not return an error code of [EINTR].

31796 EXAMPLES

31797 None.

31798 APPLICATION USAGE

31799 None.

31800 RATIONALE

31801 Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to
31802 support probable future standardization in these areas without requiring that the function itself
31803 be changed.

31804 Attributes objects provide clean isolation of the configurable aspects of threads. For example,
31805 “stack size” is an important attribute of a thread, but it cannot be expressed portably. When
31806 porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects
31807 can help by allowing the changes to be isolated in a single place, rather than being spread across

31808 every instance of thread creation.

31809 Attributes objects can be used to set up “classes” of threads with similar attributes; for example,
31810 “threads with large stacks and high priority” or “threads with minimal stacks”. These classes
31811 can be defined in a single place and then referenced wherever threads need to be created.
31812 Changes to “class” decisions become straightforward, and detailed analysis of each
31813 *pthread_create()* call is not required.

31814 The attributes objects are defined as opaque types as an aid to extensibility. If these objects had
31815 been specified as structures, adding new attributes would force recompilation of all multi-
31816 threaded programs when the attributes objects are extended; this might not be possible if
31817 different program components were supplied by different vendors.

31818 Additionally, opaque attributes objects present opportunities for improving performance.
31819 Argument validity can be checked once when attributes are set, rather than each time a thread is
31820 created. Implementations often need to cache kernel objects that are expensive to create.
31821 Opaque attributes objects provide an efficient mechanism to detect when cached objects become
31822 invalid due to attribute changes.

31823 Since assignment is not necessarily defined on a given opaque type, implementation-defined
31824 default values cannot be defined in a portable way. The solution to this problem is to allow
31825 attributes objects to be initialized dynamically by attributes object initialization functions, so
31826 that default values can be supplied automatically by the implementation.

31827 The following proposal was provided as a suggested alternative to the supplied attributes:

- 31828 1. Maintain the style of passing a parameter formed by the bitwise-inclusive OR of flags to
31829 the initialization routines (*pthread_create()*, *pthread_mutex_init()*, *pthread_cond_init()*). The
31830 parameter containing the flags should be an opaque type for extensibility. If no flags are
31831 set in the parameter, then the objects are created with default characteristics. An
31832 implementation may specify implementation-defined flag values and associated behavior.
- 31833 2. If further specialization of mutexes and condition variables is necessary, implementations
31834 may specify additional procedures that operate on the **pthread_mutex_t** and
31835 **pthread_cond_t** objects (instead of on attributes objects).

31836 The difficulties with this solution are:

- 31837 1. A bitmask is not opaque if bits have to be set into bitvector attributes objects using
31838 explicitly-coded bitwise-inclusive OR operations. If the set of options exceeds an **int**,
31839 application programmers need to know the location of each bit. If bits are set or read by
31840 encapsulation (that is, get and set functions), then the bitmask is merely an
31841 implementation of attributes objects as currently defined and should not be exposed to the
31842 programmer.
- 31843 2. Many attributes are not Boolean or very small integral values. For example, scheduling
31844 policy may be placed in 3-bit or 4-bit, but priority requires 5-bit or more, thereby taking up
31845 at least 8 bits out of a possible 16 bits on machines with 16-bit integers. Because of this, the
31846 bitmask can only reasonably control whether particular attributes are set or not, and it
31847 cannot serve as the repository of the value itself. The value needs to be specified as a
31848 function parameter (which is non-extensible), or by setting a structure field (which is non-
31849 opaque), or by get and set functions (making the bitmask a redundant addition to the
31850 attributes objects).

31851 Stack size is defined as an optional attribute because the very notion of a stack is inherently
31852 machine-dependent. Some implementations may not be able to change the size of the stack, for
31853 example, and others may not need to because stack pages may be discontiguous and can be
31854 allocated and released on demand.

31855 The attribute mechanism has been designed in large measure for extensibility. Future extensions
31856 to the attribute mechanism or to any attributes object defined in this volume of
31857 IEEE Std 1003.1-2001 has to be done with care so as not to affect binary-compatibility.

31858 Attributes objects, even if allocated by means of dynamic allocation functions such as *malloc()*,
31859 may have their size fixed at compile time. This means, for example, a *pthread_create()* in an
31860 implementation with extensions to **pthread_attr_t** cannot look beyond the area that the binary
31861 application assumes is valid. This suggests that implementations should maintain a size field in
31862 the attributes object, as well as possibly version information, if extensions in different directions
31863 (possibly by different vendors) are to be accommodated.

31864 FUTURE DIRECTIONS

31865 None.

31866 SEE ALSO

31867 *pthread_attr_getstackaddr()*, *pthread_attr_getstacksize()*, *pthread_attr_getdetachstate()*,
31868 *pthread_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

31869 CHANGE HISTORY

31870 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31871 Issue 6

31872 The *pthread_attr_destroy()* and *pthread_attr_init()* functions are marked as part of the Threads
31873 option.

31874 IEEE PASC Interpretation 1003.1 #107 is applied, noting that the effect of initializing an already
31875 initialized thread attributes object is undefined.

31876 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/71 is applied, updating the ERRORS 2
31877 section to add the optional [EINVAL] error for the *pthread_attr_destroy()* function, and the 2
31878 optional [EBUSY] error for the *pthread_attr_init()* function. 2

31879 NAME

31880 pthread_attr_getdetachstate, pthread_attr_setdetachstate — get and set the detachstate attribute

31881 SYNOPSIS

```
31882 THR #include <pthread.h>
31883     int pthread_attr_getdetachstate(const pthread_attr_t *attr,
31884             int *detachstate);
31885     int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

31887 DESCRIPTION

31888 The *detachstate* attribute controls whether the thread is created in a detached state. If the thread
 31889 is created detached, then use of the ID of the newly created thread by the *pthread_detach()* or
 31890 *pthread_join()* function is an error.

31891 The *pthread_attr_getdetachstate()* and *pthread_attr_setdetachstate()* functions, respectively, shall
 31892 get and set the *detachstate* attribute in the *attr* object.

31893 For *pthread_attr_getdetachstate()*, *detachstate* shall be set to either
 31894 PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

31895 For *pthread_attr_setdetachstate()*, the application shall set *detachstate* to either
 31896 PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

31897 A value of PTHREAD_CREATE_DETACHED shall cause all threads created with *attr* to be in
 31898 the detached state, whereas using a value of PTHREAD_CREATE_JOINABLE shall cause all
 31899 threads created with *attr* to be in the joinable state. The default value of the *detachstate* attribute
 31900 shall be PTHREAD_CREATE_JOINABLE.

31901 RETURN VALUE

31902 Upon successful completion, *pthread_attr_getdetachstate()* and *pthread_attr_setdetachstate()* shall
 31903 return a value of 0; otherwise, an error number shall be returned to indicate the error.

31904 The *pthread_attr_getdetachstate()* function stores the value of the *detachstate* attribute in *detachstate*
 31905 if successful.

31906 ERRORS

31907 The *pthread_attr_setdetachstate()* function shall fail if:

31908 [EINVAL] The value of *detachstate* was not valid

31909 These functions may fail if:

2

31910 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
 31911 object.

2
2

31912 These functions shall not return an error code of [EINTR].

31913 EXAMPLES

31914 Retrieving the detachstate Attribute

2

31915 This example shows how to obtain the *detachstate* attribute of a thread attribute object.

2

```
31916 #include <pthread.h>
```

2

```
31917 pthread_attr_t thread_attr;
```

2

```
31918 int detachstate;
```

2

```
31919 int rc;
```

2

```
31920     /* code initializing thread_attr */          2
31921     ...
31922     rc = pthread_attr_getdetachstate (&thread_attr, &detachstate);    2
31923     if (rc!=0) {                                2
31924         /* handle error */                      2
31925         ...
31926     }                                              2
31927     else {                                     2
31928         /* legal values for detachstate are:      2
31929             * PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE 2
31930             */                                2
31931         ...                                    2
31932     }                                              2
```

31933 APPLICATION USAGE

31934 None.

31935 RATIONALE

31936 None.

31937 FUTURE DIRECTIONS

31938 None.

31939 SEE ALSO

31940 *pthread_attr_destroy()*, *pthread_attr_getstackaddr()*, *pthread_attr_getstacksize()*, *pthread_create()*, the
31941 Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

31942 CHANGE HISTORY

31943 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

31944 Issue 6

31945 The *pthread_attr_setdetachstate()* and *pthread_attr_getdetachstate()* functions are marked as part of
31946 the Threads option.

31947 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

31948 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/72 is applied, adding the example to the 2
31949 EXAMPLES section. 2

31950 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/73 is applied, updating the ERRORS 2
31951 section to include the optional [EINVAL] error. 2

31952 **NAME**

31953 pthread_attr_getguardsize, pthread_attr_setguardsize — get and set the thread guardsize
 31954 attribute

31955 **SYNOPSIS**

31956 XSI #include <pthread.h>
 31957 int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
 31958 size_t *restrict guardsize);
 31959 int pthread_attr_setguardsize(pthread_attr_t *attr,
 31960 size_t guardsize);
 31961

31962 **DESCRIPTION**

31963 The *pthread_attr_getguardsize()* function shall get the *guardsize* attribute in the *attr* object. This
 31964 attribute shall be returned in the *guardsize* parameter.

31965 The *pthread_attr_setguardsize()* function shall set the *guardsize* attribute in the *attr* object. The new
 31966 value of this attribute shall be obtained from the *guardsize* parameter. If *guardsize* is zero, a guard
 31967 area shall not be provided for threads created with *attr*. If *guardsize* is greater than zero, a guard
 31968 area of at least size *guardsize* bytes shall be provided for each thread created with *attr*.

31969 The *guardsize* attribute controls the size of the guard area for the created thread's stack. The
 31970 *guardsize* attribute provides protection against overflow of the stack pointer. If a thread's stack is
 31971 created with guard protection, the implementation allocates extra memory at the overflow end
 31972 of the stack as a buffer against stack overflow of the stack pointer. If an application overflows
 31973 into this buffer an error shall result (possibly in a SIGSEGV signal being delivered to the thread).

31974 A conforming implementation may round up the value contained in *guardsize* to a multiple of
 31975 the configurable system variable {PAGESIZE} (see <sys/mman.h>). If an implementation
 31976 rounds up the value of *guardsize* to a multiple of {PAGESIZE}, a call to *pthread_attr_getguardsize()*
 31977 specifying *attr* shall store in the *guardsize* parameter the guard size specified by the previous
 31978 *pthread_attr_setguardsize()* function call.

31979 The default value of the *guardsize* attribute is {PAGESIZE} bytes. The actual value of {PAGESIZE}
 31980 is implementation-defined.

31981 If the *stackaddr* or *stack* attribute has been set (that is, the caller is allocating and managing its
 31982 own thread stacks), the *guardsize* attribute shall be ignored and no protection shall be provided
 31983 by the implementation. It is the responsibility of the application to manage stack overflow along
 31984 with stack allocation and management in this case.

31985 **RETURN VALUE**

31986 If successful, the *pthread_attr_getguardsize()* and *pthread_attr_setguardsize()* functions shall return
 31987 zero; otherwise, an error number shall be returned to indicate the error.

31988 **ERRORS**

31989 These functions shall fail if:

31990 [EINVAL] The parameter *guardsize* is invalid. 2

31991 These functions may fail if:

31992 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute 2
 31993 object. 2

31994 These functions shall not return an error code of [EINTR].

31995 EXAMPLES

31996	Retrieving the guardsize Attribute	2
31997	This example shows how to obtain the <i>guardsize</i> attribute of a thread attribute object.	2
31998	#include <pthread.h>	2
31999	pthread_attr_t thread_attr;	2
32000	size_t guardsize;	2
32001	int rc;	2
32002	/* code initializing thread_attr */	2
32003	...	2
32004	rc = pthread_attr_getguardsize (&thread_attr, &guardsize);	2
32005	if (rc != 0) {	2
32006	/* handle error */	2
32007	...	2
32008	}	2
32009	else {	2
32010	if (guardsize > 0) {	2
32011	/* a guard area of at least guardsize bytes is provided */	2
32012	...	2
32013	}	2
32014	else {	2
32015	/* no guard area provided */	2
32016	...	2
32017	}	2
32018	}	2

32019 APPLICATION USAGE

32020 None.

32021 RATIONALE

32022 The *guardsize* attribute is provided to the application for two reasons:

- 32023 1. Overflow protection can potentially result in wasted system resources. An application
32024 that creates a large number of threads, and which knows its threads never overflow their
32025 stack, can save system resources by turning off guard areas.
- 32026 2. When threads allocate large data structures on the stack, large guard areas may be needed
32027 to detect stack overflow.

32028 FUTURE DIRECTIONS

32029 None.

32030 SEE ALSO

32031 The Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>, <sys/mman.h>

32032 CHANGE HISTORY

32033 First released in Issue 5.

32034 Issue 6

32035 In the ERRORS section, a third [EINVAL] error condition is removed as it is covered by the
32036 second error condition.

32037 The **restrict** keyword is added to the *pthread_attr_getguardsize()* prototype for alignment with the
32038 ISO/IEC 9899:1999 standard.

32039	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/74 is applied, updating the ERRORS section to remove the [EINVAL] error ("The attribute attr is invalid."), and replacing it with the optional [EINVAL] error.	2
32040		2
32041		2
32042	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/76 is applied, adding the example to the EXAMPLES section.	2
32043		2

32044 NAME

32045 `pthread_attr_getinheritsched`, `pthread_attr_setinheritsched` — get and set the inheritsched
32046 attribute (**REALTIME THREADS**)

32047 SYNOPSIS

32048 THR TPS #include <pthread.h>

```
32049     int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,  
32050             int *restrict inheritsched);  
32051     int pthread_attr_setinheritsched(pthread_attr_t *attr,  
32052             int inheritsched);  
32053
```

32054 DESCRIPTION

32055 The *pthread_attr_getinheritsched()*, and *pthread_attr_setinheritsched()* functions, respectively, shall
32056 get and set the *inheritsched* attribute in the *attr* argument.

32057 When the attributes objects are used by `pthread_create()`, the `inheritsched` attribute determines
32058 how the other scheduling attributes of the created thread shall be set.

32059 The supported values of *inheritsched* shall be:

2

PTHREAD_INHERIT_SCHED

32061 Specifies that the thread scheduling attributes shall be inherited from the creating thread,
32062 and the scheduling attributes in this *attr* argument shall be ignored.

PTHREAD_EXPLICIT_SCHED

Specifies that the thread scheduling attributes shall be set to the corresponding values from this attributes object.

32066 The symbols PTHREAD_INHERIT_SCHED and PTHREAD_EXPLICIT_SCHED are defined in
32067 the <pthread.h> header.

32068 The following thread scheduling attributes defined by IEEE Std 1003.1-2001 are affected by the
32069 *inheritsched* attribute: scheduling policy (*schedpolicy*), scheduling parameters (*schedparam*), and
32070 scheduling contention scope (*contentionscope*).

32071 RETURN VALUE

If successful, the *pthread_attr_getinheritsched()* and *pthread_attr_setinheritsched()* functions shall return zero; otherwise, an error number shall be returned to indicate the error.

32074 ERRORS

32075 The *pthread_attr_getinheritsched()* function may fail if:

2

32076 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.
32077

32078 The *pthread_attr_setinheritsched()* function may fail if:

32079 [EINVAL] The value of *inheritsched* is not valid.

32080 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.
32081

32082 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

32083 These functions shall not return an error code of [EINTR].

32084 EXAMPLES

32085 None.

32086 APPLICATION USAGE

32087 After these attributes have been set, a thread can be created with the specified attributes using
32088 *pthread_create()*. Using these routines does not affect the current running thread.

32089 See Section 2.9.4 (on page 52) for further details on thread scheduling attributes and their default 2
32090 settings. 2

32091 RATIONALE

32092 None.

32093 FUTURE DIRECTIONS

32094 None.

32095 SEE ALSO

32096 *pthread_attr_destroy()*, *pthread_attr_getscope()*, *pthread_attr_getschedpolicy()*,
32097 *pthread_attr_getschedparam()*, *pthread_create()*, the Base Definitions volume of
32098 IEEE Std 1003.1-2001, <*pthread.h*>, <*sched.h*>

32099 CHANGE HISTORY

32100 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32101 Marked as part of the Realtime Threads Feature Group.

32102 Issue 6

32103 The *pthread_attr_getinheritsched()* and *pthread_attr_setinheritsched()* functions are marked as part
32104 of the Threads and Thread Execution Scheduling options.

32105 The [ENOSYS] error condition has been removed as stubs need not be provided if an 2
32106 implementation does not support the Thread Execution Scheduling option. 2

32107 The **restrict** keyword is added to the *pthread_attr_getinheritsched()* prototype for alignment with 2
32108 the ISO/IEC 9899: 1999 standard. 2

32109 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/75 is applied, clarifying the values of 2
32110 *inheritsched* in the DESCRIPTION and adding two optional [EINVAL] errors to the ERRORS 2
32111 section for checking when *attr* refers to an uninitialized thread attribute object. 2

32112 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/77 is applied, adding a reference to 2
32113 Section 2.9.4 (on page 52) in the APPLICATION USAGE section. 2

32114 NAME

32115 pthread_attr_getschedparam, pthread_attr_setschedparam — get and set the schedparam
 32116 attribute

32117 SYNOPSIS

32118 THR #include <pthread.h>

```
32119        int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
32120                struct sched_param *restrict param);
32121        int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
32122                const struct sched_param *restrict param);
```

32123

32124 DESCRIPTION

32125 The *pthread_attr_getschedparam()*, and *pthread_attr_setschedparam()* functions, respectively, shall
 32126 get and set the scheduling parameter attributes in the *attr* argument. The contents of the *param*
 32127 structure are defined in the <sched.h> header. For the SCED_FIFO and SCED_RR policies,
 32128 the only required member of *param* is *sched_priority*.

32129 TSP For the SCED_SPORADIC policy, the required members of the *param* structure are
 32130 *sched_priority*, *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and
 32131 *sched_ss_max_repl*. The specified *sched_ss_repl_period* must be greater than or equal to the
 32132 specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.
 32133 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
 32134 function to succeed; if not, the function shall fail.

32135 RETURN VALUE

32136 If successful, the *pthread_attr_getschedparam()* and *pthread_attr_setschedparam()* functions shall
 32137 return zero; otherwise, an error number shall be returned to indicate the error.

32138 ERRORS

32139	The <i>pthread_attr_getschedparam()</i> function may fail if:	2
32140	[EINVAL] The value specified by <i>attr</i> does not refer to an initialized thread attribute	2
32141	object.	2
32142	The <i>pthread_attr_setschedparam()</i> function may fail if:	
32143	[EINVAL] The value of <i>param</i> is not valid, or the value specified by <i>attr</i> does not refer to	2
32144	an initialized thread attribute object.	2
32145	[ENOTSUP] An attempt was made to set the attribute to an unsupported value.	
32146	These functions shall not return an error code of [EINTR].	

32147 EXAMPLES

32148 None.

32149 APPLICATION USAGE

32150 After these attributes have been set, a thread can be created with the specified attributes using
 32151 *pthread_create()*. Using these routines does not affect the current running thread.

32152 RATIONALE

32153 None.

32154 FUTURE DIRECTIONS

32155 None.

32156 SEE ALSO

32157 *pthread_attr_destroy()*, *pthread_attr_getscope()*, *pthread_attr_getinheritsched()*,
32158 *pthread_attr_getschedpolicy()*, *pthread_create()*, the Base Definitions volume of
32159 IEEE Std 1003.1-2001, <**pthread.h**>, <**sched.h**>

32160 CHANGE HISTORY

32161 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32162 Issue 6

32163 The *pthread_attr_getschedparam()* and *pthread_attr_setschedparam()* functions are marked as part
32164 of the Threads option.

32165 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

32166 The **restrict** keyword is added to the *pthread_attr_getschedparam()* and
32167 *pthread_attr_setschedparam()* prototypes for alignment with the ISO/IEC 9899: 1999 standard.

32168 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/78 is applied, updating the ERRORS 2
32169 section to include optional errors for the case when *attr* refers to an uninitialized thread attribute 2
32170 object. 2

32171 NAME

32172 `pthread_attr_getschedpolicy`, `pthread_attr_setschedpolicy` — get and set the `schedpolicy`
32173 attribute (**REALTIME THREADS**)

32174 SYNOPSIS

32175 THR TPS #include <pthread.h>

```
32176     int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,  
32177             int *restrict policy);  
32178     int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);  
32179
```

32180 DESCRIPTION

32181 The *pthread_attr_getschedpolicy()* and *pthread_attr_setschedpolicy()* functions, respectively, shall
32182 get and set the *schedpolicy* attribute in the *attr* argument.

32183 The supported values of *policy* shall include SCHED_FIFO, SCHED_RR, and SCHED_OTHER,
32184 which are defined in the <sched.h> header. When threads executing with the scheduling policy
32185 TSP SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC are waiting on a mutex, they shall acquire
32186 the mutex in priority order when the mutex is unlocked.

32187 RETURN VALUE

If successful, the *pthread_attr_getschedpolicy()* and *pthread_attr_setschedpolicy()* functions shall return zero; otherwise, an error number shall be returned to indicate the error.

32190 ERRORS

32191 The *pthread_attr_getschedpolicy()* function may fail if:

32192 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.
32193

32194 The *pthread_attr_setschedpolicy()* function may fail if:

32195 [EINVAL] The value of *policy* is not valid, or the value specified by *attr* does not refer to 2
32196 an initialized thread attribute object. 2

32197 [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

32198 These functions shall not return an error code of [EINTR].

32199 EXAMPLES

32200 None.

32201 APPLICATION USAGE

32202 After these attributes have been set, a thread can be created with the specified attributes using
32203 *pthread_create()*. Using these routines does not affect the current running thread.

32204 See Section 2.9.4 (on page 52) for further details on thread scheduling attributes and their default settings. 2
32205 2

32206 RATIONALE

32207 None.

32208 FUTURE DIRECTIONS

32209 None.

32210 SEE ALSO

32211 *pthread_attr_destroy()*, *pthread_attr_getscope()*, *pthread_attr_getinheritsched()*,
32212 *pthread_attr_getschedparam()*, *pthread_create()*, the Base Definitions volume of
32213 IEEE Std 1003.1-2001, <*pthread.h*>, <*sched.h*>

32214 CHANGE HISTORY

32215 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32216 Marked as part of the Realtime Threads Feature Group.

32217 Issue 6

32218 The *pthread_attr_getschedpolicy()* and *pthread_attr_setschedpolicy()* functions are marked as part of
32219 the Threads and Thread Execution Scheduling options.

32220 The [ENOSYS] error condition has been removed as stubs need not be provided if an
32221 implementation does not support the Thread Execution Scheduling option.

32222 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

32223 The **restrict** keyword is added to the *pthread_attr_getschedpolicy()* prototype for alignment with
32224 the ISO/IEC 9899:1999 standard.

32225 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/79 is applied, adding a reference to 2
32226 Section 2.9.4 (on page 52) in the APPLICATION USAGE section. 2

32227 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/80 is applied, updating the ERRORS 2
32228 section to include optional errors for the case when *attr* refers to an uninitialized thread attribute 2
32229 object. 2

32230 NAME

32231 `pthread_attr_getscope`, `pthread_attr_setscope` — get and set the `contentionscope` attribute
 32232 (**REALTIME THREADS**)

32233 SYNOPSIS

```
32234 THR TPS #include <pthread.h>
32235     int pthread_attr_getscope(const pthread_attr_t *restrict attr,
32236         int *restrict contentionscope);
32237     int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
32238
```

32239 DESCRIPTION

32240 The `pthread_attr_getscope()` and `pthread_attr_setscope()` functions, respectively, shall get and set
 32241 the `contentionscope` attribute in the `attr` object.

32242 The `contentionscope` attribute may have the values `PTHREAD_SCOPE_SYSTEM`, signifying
 32243 system scheduling contention scope, or `PTHREAD_SCOPE_PROCESS`, signifying process
 32244 scheduling contention scope. The symbols `PTHREAD_SCOPE_SYSTEM` and
 32245 `PTHREAD_SCOPE_PROCESS` are defined in the `<pthread.h>` header.

32246 RETURN VALUE

32247 If successful, the `pthread_attr_getscope()` and `pthread_attr_setscope()` functions shall return zero;
 32248 otherwise, an error number shall be returned to indicate the error.

32249 ERRORS

32250	The <code>pthread_attr_getscope()</code> function may fail if:	2
32251	[EINVAL] The value specified by <code>attr</code> does not refer to an initialized thread attribute	2
32252	object.	2
32253	The <code>pthread_attr_setscope()</code> function may fail if:	
32254	[EINVAL] The value of <code>contentionscope</code> is not valid, or the value specified by <code>attr</code> does not	2
32255	refer to an initialized thread attribute object.	2
32256	[ENOTSUP] An attempt was made to set the attribute to an unsupported value.	
32257	These functions shall not return an error code of [EINTR].	

32258 EXAMPLES

32259 None.

32260 APPLICATION USAGE

32261 After these attributes have been set, a thread can be created with the specified attributes using
 32262 `pthread_create()`. Using these routines does not affect the current running thread.

32263 See Section 2.9.4 (on page 52) for further details on thread scheduling attributes and their default 2
 32264 settings.

32265 RATIONALE

32266 None.

32267 FUTURE DIRECTIONS

32268 None.

32269 SEE ALSO

32270 `pthread_attr_destroy()`, `pthread_attr_getinheritsched()`, `pthread_attr_getschedpolicy()`,
 32271 `pthread_attr_getschedparam()`, `pthread_create()`, the Base Definitions volume of
 32272 IEEE Std 1003.1-2001, `<pthread.h>`, `<sched.h>`

32273 CHANGE HISTORY

32274 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32275 Marked as part of the Realtime Threads Feature Group.

32276 Issue 6

32277 The *pthread_attr_getscope()* and *pthread_attr_setscope()* functions are marked as part of the
32278 Threads and Thread Execution Scheduling options.

32279 The [ENOSYS] error condition has been removed as stubs need not be provided if an
32280 implementation does not support the Thread Execution Scheduling option.

32281 The **restrict** keyword is added to the *pthread_attr_getscope()* prototype for alignment with the
32282 ISO/IEC 9899:1999 standard.

32283 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/81 is applied, adding a reference to 2
32284 Section 2.9.4 (on page 52) in the APPLICATION USAGE section. 2

32285 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/82 is applied, updating the ERRORS 2
32286 section to include optional errors for the case when *attr* refers to an uninitialized thread attribute 2
32287 object. 2

32288 NAME

32289 *pthread_attr_getstack*, *pthread_attr_setstack* — get and set stack attributes

32290 SYNOPSIS

32291 THR *#include <pthread.h>*

32292 TSA TSS *int pthread_attr_getstack(const pthread_attr_t *restrict attr,*
32293 *void **restrict stackaddr, size_t *restrict stacksize);*
32294 *int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,*
32295 *size_t stacksize);*

32296

32297 DESCRIPTION

32298 The *pthread_attr_getstack()* and *pthread_attr_setstack()* functions, respectively, shall get and set
32299 the thread creation stack attributes *stackaddr* and *stacksize* in the *attr* object.

32300 The stack attributes specify the area of storage to be used for the created thread's stack. The base
32301 (lowest addressable byte) of the storage shall be *stackaddr*, and the size of the storage shall be
32302 *stacksize* bytes. The *stacksize* shall be at least {PTHREAD_STACK_MIN}. The *stackaddr* shall be
32303 aligned appropriately to be used as a stack; for example, *pthread_attr_setstack()* may fail with
32304 [EINVAL] if (*stackaddr* & 0x7) is not 0. All pages within the stack described by *stackaddr* and
32305 *stacksize* shall be both readable and writable by the thread.

32306 RETURN VALUE

32307 Upon successful completion, these functions shall return a value of 0; otherwise, an error
32308 number shall be returned to indicate the error.

32309 The *pthread_attr_getstack()* function shall store the stack attribute values in *stackaddr* and *stacksize*
32310 if successful.

32311 ERRORS

32312 The *pthread_attr_setstack()* function shall fail if:

32313 [EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds an
32314 implementation-defined limit.

32315 The *pthread_attr_getstack()* function may fail if:

2

32316 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
32317 object.

2

32318 The *pthread_attr_setstack()* function may fail if:

32319 [EINVAL] The value of *stackaddr* does not have proper alignment to be used as a stack, or
32320 (*stackaddr* + *stacksize*) lacks proper alignment, or the value specified by *attr*
32321 does not refer to an initialized thread attribute object.

2

32322 [EACCES] The stack page(s) described by *stackaddr* and *stacksize* are not both readable
32323 and writable by the thread.

32324 These functions shall not return an error code of [EINTR].

32325 EXAMPLES

32326 None.

32327 APPLICATION USAGE

32328 These functions are appropriate for use by applications in an environment where the stack for a
32329 thread must be placed in some particular region of memory.

32330 While it might seem that an application could detect stack overflow by providing a protected
32331 page outside the specified stack region, this cannot be done portably. Implementations are free
32332 to place the thread's initial stack pointer anywhere within the specified region to accommodate
32333 the machine's stack pointer behavior and allocation requirements. Furthermore, on some
32334 architectures, such as the IA-64, "overflow" might mean that two separate stack pointers
32335 allocated within the region will overlap somewhere in the middle of the region.

32336 After a successful call to *pthread_attr_setstack()*, the storage area specified by the *stackaddr* 2
32337 parameter is under the control of the implementation, as described in Section 2.9.8 (on page 58). 2

32338 RATIONALE

32339 None.

32340 FUTURE DIRECTIONS

32341 None.

32342 SEE ALSO

32343 *pthread_attr_init()*, *pthread_attr_setdetachstate()*, *pthread_attr_setstacksize()*, *pthread_create()*, the
32344 Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <pthread.h>

32345 CHANGE HISTORY

32346 First released in Issue 6. Developed as an XSI extension and brought into the BASE by IEEE
32347 PASC Interpretation 1003.1 #101.

32348 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/83 is applied, updating the 2
32349 APPLICATION USAGE section to refer to Section 2.9.8 (on page 58). 2

32350 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC/D6/84 is applied, updating the ERRORS 2
32351 section to include optional errors for the case when *attr* refers to an uninitialized thread attribute
32352 object. 2

32353 NAME

`pthread_attr_getstackaddr`, `pthread_attr_setstackaddr` — get and set the stackaddr attribute

32355 SYNOPSIS

32356 THR TSA #include <pthread.h>

```
32357 OB     int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,  
32358             void **restrict stackaddr);  
32359     int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);  
32360
```

32361 DESCRIPTION

32362 The *pthread_attr_getstackaddr()* and *pthread_attr_setstackaddr()* functions, respectively, shall get
32363 and set the thread creation *stackaddr* attribute in the *attr* object.

32364 The *stackaddr* attribute specifies the location of storage to be used for the created thread's stack.
32365 The size of the storage shall be at least {PTHREAD_STACK_MIN}.

32366 RETURN VALUE

32367 Upon successful completion, *pthread_attr_getstackaddr()* and *pthread_attr_setstackaddr()* shall
32368 return a value of 0; otherwise, an error number shall be returned to indicate the error.

32369 The `pthread_attr_getstackaddr()` function stores the `stackaddr` attribute value in `stackaddr` if
32370 successful.

32371 ERRORS

32372 These functions may fail if:

2

32373 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute object.
32374

32375 These functions shall not return an error code of [EINTR].

32376 EXAMPLES

32377 None.

32378 APPLICATION USAGE

The specification of the *stackaddr* attribute presents several ambiguities that make portable use of these interfaces impossible. The description of the single address parameter as a “stack” does not specify a particular relationship between the address and the “stack” implied by that address. For example, the address may be taken as the low memory address of a buffer intended for use as a stack, or it may be taken as the address to be used as the initial stack pointer register value for the new thread. These two are not the same except for a machine on which the stack grows “up” from low memory to high, and on which a “push” operation first stores the value in memory and then increments the stack pointer register. Further, on a machine where the stack grows “down” from high memory to low, interpretation of the address as the “low memory” address requires a determination of the intended size of the stack. IEEE Std 1003.1-2001 has introduced the new interfaces *pthread_attr_setstack()* and *pthread_attr_getstack()* to resolve these ambiguities.

32391 After a successful call to `pthread_attr_setstackaddr()`, the storage area specified by the `stackaddr` 22
32392 parameter is under the control of the implementation, as described in Section 2.9.8 (on page 58). 22

32393 RATIONALE

32394 None.

32395 **FUTURE DIRECTIONS**

32396 None.

32397 **SEE ALSO**

32398 *pthread_attr_destroy()*, *pthread_attr_getdetachstate()*, *pthread_attr_getstack()*,
32399 *pthread_attr_getstacksize()*, *pthread_attr_setstack()*, *pthread_create()*, the Base Definitions volume
32400 of IEEE Std 1003.1-2001, <limits.h>, <pthread.h>

32401 **CHANGE HISTORY**

32402 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32403 **Issue 6**

32404 The *pthread_attr_getstackaddr()* and *pthread_attr_setstackaddr()* functions are marked as part of
32405 the Threads and Thread Stack Address Attribute options.

32406 The **restrict** keyword is added to the *pthread_attr_getstackaddr()* prototype for alignment with the
32407 ISO/IEC 9899: 1999 standard.

32408 These functions are marked obsolescent.

32409 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/85 is applied, updating the 2
32410 APPLICATION USAGE section to refer to Section 2.9.8 (on page 58). 2

32411 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/86 is applied, updating the ERRORS 2
32412 section to include optional errors for the case when *attr* refers to an uninitialized thread attribute 2
32413 object. 2

32414 NAME

32415 *pthread_attr_getstacksize*, *pthread_attr_setstacksize* — get and set the stacksize attribute

32416 SYNOPSIS

32417 THR TSS #include <pthread.h>

```
32418     int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
32419             size_t *restrict stacksize);
32420     int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

32422 DESCRIPTION

32423 The *pthread_attr_getstacksize()* and *pthread_attr_setstacksize()* functions, respectively, shall get
32424 and set the thread creation *stacksize* attribute in the *attr* object.

32425 The *stacksize* attribute shall define the minimum stack size (in bytes) allocated for the created
32426 threads stack.

32427 RETURN VALUE

32428 Upon successful completion, *pthread_attr_getstacksize()* and *pthread_attr_setstacksize()* shall
32429 return a value of 0; otherwise, an error number shall be returned to indicate the error.

32430 The *pthread_attr_getstacksize()* function stores the *stacksize* attribute value in *stacksize* if
32431 successful.

32432 ERRORS

32433 The *pthread_attr_setstacksize()* function shall fail if:

32434 [EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds a
32435 system-imposed limit.

32436 These functions may fail if:

32437 [EINVAL] The value specified by *attr* does not refer to an initialized thread attribute
32438 object.

32439 These functions shall not return an error code of [EINTR].

32440 EXAMPLES

32441 None.

32442 APPLICATION USAGE

32443 None.

32444 RATIONALE

32445 None.

32446 FUTURE DIRECTIONS

32447 None.

32448 SEE ALSO

32449 *pthread_attr_destroy()*, *pthread_attr_getstackaddr()*, *pthread_attr_getdetachstate()*, *pthread_create()*,
32450 the Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <pthread.h>

32451 CHANGE HISTORY

32452 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32453 Issue 6

32454 The *pthread_attr_getstacksize()* and *pthread_attr_setstacksize()* functions are marked as part of the
32455 Threads and Thread Stack Size Attribute options.

32456	The restrict keyword is added to the <i>pthread_attr_getstacksize()</i> prototype for alignment with the ISO/IEC 9899:1999 standard.	
32458	IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/43 is applied, correcting the margin code in the SYNOPSIS from TSA to TSS and updating the CHANGE HISTORY from “Thread Stack Address Attribute” option to “Thread Stack Size Attribute” option.	1
32461	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/87 is applied, updating the ERRORS section to include optional errors for the case when <i>attr</i> refers to an uninitialized thread attribute object.	2
32462		2
32463		2

32464 **NAME**

32465 pthread_attr_init — initialize the thread attributes object

32466 **SYNOPSIS**

32467 **THR** #include <pthread.h>

32468 int pthread_attr_init(pthread_attr_t *attr);

32469

32470 **DESCRIPTION**

32471 Refer to *pthread_attr_destroy()*.

32472 NAME

32473 `pthread_attr_setdetachstate` — set the detachstate attribute

32474 SYNOPSIS

32475 THR `#include <pthread.h>`

32476 `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);`

32477

32478 DESCRIPTION

32479 Refer to *pthread_attr_getdetachstate()*.

32480 NAME

32481 `pthread_attr_setguardsize` — set the thread guardsize attribute

32482 SYNOPSIS

32483 XSI `#include <pthread.h>`

32484 `int pthread_attr_setguardsize(pthread_attr_t *attr,`
32485 `size_t guardsize);`

32486

32487 DESCRIPTION

32488 Refer to `pthread_attr_getguardsize()`.

32489 NAME

32490 **pthread_attr_setinheritsched** — set the inheritsched attribute (**REALTIME THREADS**)

32491 SYNOPSIS

32492 THR TPS #include <pthread.h>

```
32493     int pthread_attr_setinheritsched(pthread_attr_t *attr,  
32494         int inheritsched);
```

32495

32496 DESCRIPTION

32497 Refer to *pthread_attr_getinheritsched()*.

32498 **NAME**32499 **pthread_attr_setschedparam** — set the schedparam attribute32500 **SYNOPSIS**32501 **THR** `#include <pthread.h>`32502 `int pthread_attr_setschedparam(pthread_attr_t *restrict attr,`
32503 `const struct sched_param *restrict param);`

32504

32505 **DESCRIPTION**32506 Refer to *pthread_attr_getschedparam()*.

32507 NAME

32508 `pthread_attr_setschedpolicy` — set the schedpolicy attribute (**REALTIME THREADS**)

32509 SYNOPSIS

32510 THR TPS `#include <pthread.h>`

32511 `int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);`

32512

32513 DESCRIPTION

32514 Refer to *pthread_attr_getschedpolicy()*.

32515 **NAME**32516 `pthread_attr_setscope` — set the contentionscope attribute (**REALTIME THREADS**)32517 **SYNOPSIS**32518 **THR TPS** `#include <pthread.h>`32519 `int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);`

32520

32521 **DESCRIPTION**32522 Refer to *pthread_attr_getscope()*.

32523 NAME

32524 `pthread_attr_setstack` — set the stack attribute

32525 SYNOPSIS

32526 XSI `#include <pthread.h>`

32527 `int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,`
32528 `size_t stacksize);`

32529

32530 DESCRIPTION

32531 Refer to *pthread_attr_getstack()*.

32532 **NAME**32533 `pthread_attr_setstackaddr` — set the stackaddr attribute32534 **SYNOPSIS**32535 THR TSA `#include <pthread.h>`32536 OB `int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);`

32537

32538 **DESCRIPTION**32539 Refer to *pthread_attr_getstackaddr()*.

32540 **NAME**

32541 pthread_attr_setstacksize — set the stacksize attribute

32542 **SYNOPSIS**

32543 THR TSS #include <pthread.h>

1

32544 int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

32545

32546 **DESCRIPTION**

32547 Refer to *pthread_attr_getstacksize()*.

32548 NAME

32549 pthread_barrier_destroy, pthread_barrier_init — destroy and initialize a barrier object
 32550 (**ADVANCED REALTIME THREADS**)

32551 SYNOPSIS

```
32552 THR BAR #include <pthread.h>
32553     int pthread_barrier_destroy(pthread_barrier_t *barrier);
32554     int pthread_barrier_init(pthread_barrier_t *restrict barrier,
32555         const pthread_barrierattr_t *restrict attr, unsigned count);
```

32556

32557 DESCRIPTION

32558 The *pthread_barrier_destroy()* function shall destroy the barrier referenced by *barrier* and release
 32559 any resources used by the barrier. The effect of subsequent use of the barrier is undefined until
 32560 the barrier is reinitialized by another call to *pthread_barrier_init()*. An implementation may use
 32561 this function to set *barrier* to an invalid value. The results are undefined if
 32562 *pthread_barrier_destroy()* is called when any thread is blocked on the barrier, or if this function is
 32563 called with an uninitialized barrier.

32564 The *pthread_barrier_init()* function shall allocate any resources required to use the barrier
 32565 referenced by *barrier* and shall initialize the barrier with attributes referenced by *attr*. If *attr* is
 32566 NULL, the default barrier attributes shall be used; the effect is the same as passing the address of
 32567 a default barrier attributes object. The results are undefined if *pthread_barrier_init()* is called
 32568 when any thread is blocked on the barrier (that is, has not returned from the
 32569 *pthread_barrier_wait()* call). The results are undefined if a barrier is used without first being
 32570 initialized. The results are undefined if *pthread_barrier_init()* is called specifying an already
 32571 initialized barrier.

32572 The *count* argument specifies the number of threads that must call *pthread_barrier_wait()* before
 32573 any of them successfully return from the call. The value specified by *count* must be greater than
 32574 zero.

32575 If the *pthread_barrier_init()* function fails, the barrier shall not be initialized and the contents of
 32576 *barrier* are undefined.

32577 Only the object referenced by *barrier* may be used for performing synchronization. The result of
 32578 referring to copies of that object in calls to *pthread_barrier_destroy()* or *pthread_barrier_wait()* is
 32579 undefined.

32580 RETURN VALUE

32581 Upon successful completion, these functions shall return zero; otherwise, an error number shall
 32582 be returned to indicate the error.

32583 ERRORS

32584 The *pthread_barrier_destroy()* function may fail if:

32585 [EBUSY] The implementation has detected an attempt to destroy a barrier while it is in
 32586 use (for example, while being used in a *pthread_barrier_wait()* call) by another
 32587 thread.

32588 [EINVAL] The value specified by *barrier* is invalid.

32589 The *pthread_barrier_init()* function shall fail if:

32590 [EAGAIN] The system lacks the necessary resources to initialize another barrier.

32591 [EINVAL] The value specified by *count* is equal to zero.

32592 [ENOMEM] Insufficient memory exists to initialize the barrier.

32593 The *pthread_barrier_init()* function may fail if:

32594 [EBUSY] The implementation has detected an attempt to reinitialize a barrier while it is in use (for example, while being used in a *pthread_barrier_wait()* call) by another thread.

32597 [EINVAL] The value specified by *attr* is invalid.

32598 These functions shall not return an error code of [EINTR].

32599 EXAMPLES

32600 None.

32601 APPLICATION USAGE

32602 The *pthread_barrier_destroy()* and *pthread_barrier_init()* functions are part of the Barriers option and need not be provided on all implementations.

32604 RATIONALE

32605 None.

32606 FUTURE DIRECTIONS

32607 None.

32608 SEE ALSO

32609 *pthread_barrier_wait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

32610 CHANGE HISTORY

32611 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

32612 NAME

32613 pthread_barrier_wait — synchronize at a barrier (**ADVANCED REALTIME THREADS**)

32614 SYNOPSIS

32615 THR BAR #include <pthread.h>

```
32616        int pthread_barrier_wait(pthread_barrier_t *barrier);
```

32617

32618 DESCRIPTION

32619 The *pthread_barrier_wait()* function shall synchronize participating threads at the barrier referenced by *barrier*. The calling thread shall block until the required number of threads have called *pthread_barrier_wait()* specifying the barrier.

32622 When the required number of threads have called *pthread_barrier_wait()* specifying the barrier, the constant PTHREAD_BARRIER_SERIAL_THREAD shall be returned to one unspecified thread and zero shall be returned to each of the remaining threads. At this point, the barrier shall be reset to the state it had as a result of the most recent *pthread_barrier_init()* function that referenced it.

32627 The constant PTHREAD_BARRIER_SERIAL_THREAD is defined in <pthread.h> and its value shall be distinct from any other value returned by *pthread_barrier_wait()*.

32629 The results are undefined if this function is called with an uninitialized barrier.

32630 If a signal is delivered to a thread blocked on a barrier, upon return from the signal handler the thread shall resume waiting at the barrier if the barrier wait has not completed (that is, if the required number of threads have not arrived at the barrier during the execution of the signal handler); otherwise, the thread shall continue as normal from the completed barrier wait. Until the thread in the signal handler returns from it, it is unspecified whether other threads may proceed past the barrier once they have all reached it.

32636 A thread that has blocked on a barrier shall not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution. Eligibility for processing resources shall be determined by the scheduling policy.

32639 RETURN VALUE

32640 Upon successful completion, the *pthread_barrier_wait()* function shall return PTHREAD_BARRIER_SERIAL_THREAD for a single (arbitrary) thread synchronized at the barrier and zero for each of the other threads. Otherwise, an error number shall be returned to indicate the error.

32644 ERRORS

32645 The *pthread_barrier_wait()* function may fail if:

32646 [EINVAL] The value specified by *barrier* does not refer to an initialized barrier object.

32647 This function shall not return an error code of [EINTR].

32648 EXAMPLES

32649 None.

32650 APPLICATION USAGE

32651 Applications using this function may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

32653 The *pthread_barrier_wait()* function is part of the Barriers option and need not be provided on all implementations.

32655 RATIONALE

32656 None.

32657 FUTURE DIRECTIONS

32658 None.

32659 SEE ALSO

32660 *pthread_barrier_destroy()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

32661 CHANGE HISTORY

32662 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

32663 In the SYNOPSIS, the inclusion of <**sys/types.h**> is no longer required.

32664 NAME

32665 `pthread_barrierattr_destroy`, `pthread_barrierattr_init` — destroy and initialize the barrier
32666 attributes object (**ADVANCED REALTIME THREADS**)

32667 SYNOPSIS

```
32668 THR BAR #include <pthread.h>  
  
32669     int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);  
32670     int pthread_barrierattr_init(pthread_barrierattr_t *attr);  
32671
```

32672 DESCRIPTION

32673 The `pthread_barrierattr_destroy()` function shall destroy a barrier attributes object. A destroyed
32674 *attr* attributes object can be reinitialized using `pthread_barrierattr_init()`; the results of otherwise
32675 referencing the object after it has been destroyed are undefined. An implementation may cause
32676 `pthread_barrierattr_destroy()` to set the object referenced by *attr* to an invalid value.

32677 The `pthread_barrierattr_init()` function shall initialize a barrier attributes object *attr* with the
32678 default value for all of the attributes defined by the implementation.

32679 Results are undefined if `pthread_barrierattr_init()` is called specifying an already initialized *attr*
32680 attributes object.

32681 After a barrier attributes object has been used to initialize one or more barriers, any function
32682 affecting the attributes object (including destruction) shall not affect any previously initialized
32683 barrier.

32684 RETURN VALUE

32685 If successful, the `pthread_barrierattr_destroy()` and `pthread_barrierattr_init()` functions shall return
32686 zero; otherwise, an error number shall be returned to indicate the error.

32687 ERRORS

32688 The `pthread_barrierattr_destroy()` function may fail if:

32689 [EINVAL] The value specified by *attr* is invalid.

32690 The `pthread_barrierattr_init()` function shall fail if:

32691 [ENOMEM] Insufficient memory exists to initialize the barrier attributes object.

32692 These functions shall not return an error code of [EINTR].

32693 EXAMPLES

32694 None.

32695 APPLICATION USAGE

32696 The `pthread_barrierattr_destroy()` and `pthread_barrierattr_init()` functions are part of the Barriers
32697 option and need not be provided on all implementations.

32698 RATIONALE

32699 None.

32700 FUTURE DIRECTIONS

32701 None.

32702 SEE ALSO

32703 `pthread_barrierattr_getpshared()`, `pthread_barrierattr_setpshared()`, the Base Definitions volume of
32704 IEEE Std 1003.1-2001, <pthread.h>.

32705 CHANGE HISTORY

- 32706 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.
- 32707 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

32708 NAME

32709 `pthread_barrierattr_getpshared`, `pthread_barrierattr_setpshared` — get and set the process-shared attribute of the barrier attributes object (**ADVANCED REALTIME THREADS**)

32711 SYNOPSIS

32712 THR #include <pthread.h>

```
32713 BAR TSH int pthread_barrierattr_getpshared(const pthread_barrierattr_t *  
32714             restrict attr, int *restrict pshared);  
32715 int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,  
32716             int pshared);  
32717
```

32718 DESCRIPTION

32719 The *pthread_barrierattr_getpshared()* function shall obtain the value of the *process-shared* attribute
32720 from the attributes object referenced by *attr*. The *pthread_barrierattr_setpshared()* function shall
32721 set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a barrier to be operated upon by any thread that has access to the memory where the barrier is allocated. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the barrier shall only be operated upon by threads created within the same process as the thread that initialized the barrier; if threads of different processes attempt to operate on such a barrier, the behavior is undefined. The default value of the attribute shall be PTHREAD_PROCESS_PRIVATE. Both constants PTHREAD_PROCESS_SHARED and PTHREAD_PROCESS_PRIVATE are defined in <pthread.h>.

32730 Additional attributes, their default values, and the names of the associated functions to get and
32731 set those attribute values are implementation-defined.

32732 RETURN VALUE

If successful, the *pthread_barrierattr_getpshared()* function shall return zero and store the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_barrierattr_setpshared()* function shall return zero; otherwise, an error number shall be returned to indicate the error.

32738 ERRORS

32739 These functions may fail if:

32740 [EINVAL] The value specified by attr is invalid.

32741 The *pthread_barrierattr_setpshared()* function may fail if:

32742 [EINVAL] The new value specified for the *process-shared* attribute is not one of the legal values PTHREAD_PROCESS_SHARED or PTHREAD_PROCESS_PRIVATE.
32743

32744 These functions shall not return an error code of [EINTR].

32745 EXAMPLES

32746 None.

32747 APPLICATION USAGE

32748 The *pthread_barrierattr_getpshared()* and *pthread_barrierattr_setpshared()* functions are part of the
32749 Barriers option and need not be provided on all implementations.

32750 RATIONALE

32751 None.

32752 FUTURE DIRECTIONS

32753 None.

32754 SEE ALSO

32755 *pthread_barrier_destroy()*, *pthread_barrierattr_destroy()*, *pthread_barrierattr_init()*, the Base
32756 Definitions volume of IEEE Std 1003.1-2001, <*pthread.h*>

32757 CHANGE HISTORY

32758 First released in Issue 6. Derived from IEEE Std 1003.1j-2000

32759 NAME

32760 **pthread_barrierattr_init** — initialize the barrier attributes object (**ADVANCED REALTIME**
32761 **THREADS**)

32762 SYNOPSIS

32763 **THR BAR** #include <pthread.h>

32764 int **pthread_barrierattr_init**(**pthread_barrierattr_t** *attr);

32765

32766 DESCRIPTION

32767 Refer to *pthread_barrierattr_destroy()*.

32768 NAME

32769 **pthread_barrierattr_setpshared** — set the process-shared attribute of the barrier attributes object
32770 **(ADVANCED REALTIME THREADS)**

32771 SYNOPSIS

32772 THR

```
#include <pthread.h>
```

32773 BAR TSH int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
32774 int pshared);
32775

32776 DESCRIPTION

32777 Refer to *pthread_barrierattr_getpshared()*.

32778 NAME

32779 **pthread_cancel** — cancel execution of a thread

32780 SYNOPSIS

32781 **THR** `#include <pthread.h>`

32782 `int pthread_cancel(pthread_t thread);`

32783

32784 DESCRIPTION

32785 The *pthread_cancel()* function shall request that *thread* be canceled. The target thread's
32786 cancelability state and type determines when the cancellation takes effect. When the cancellation
32787 is acted on, the cancellation cleanup handlers for *thread* shall be called. When the last
32788 cancellation cleanup handler returns, the thread-specific data destructor functions shall be called
32789 for *thread*. When the last destructor function returns, *thread* shall be terminated.

32790 The cancellation processing in the target thread shall run asynchronously with respect to the
32791 calling thread returning from *pthread_cancel()*.

32792 RETURN VALUE

32793 If successful, the *pthread_cancel()* function shall return zero; otherwise, an error number shall be
32794 returned to indicate the error.

32795 ERRORS

32796 The *pthread_cancel()* function may fail if:

32797 [ESRCH] No thread could be found corresponding to that specified by the given thread
32798 ID.

32799 The *pthread_cancel()* function shall not return an error code of [EINTR].

32800 EXAMPLES

32801 None.

32802 APPLICATION USAGE

32803 None.

32804 RATIONALE

32805 Two alternative functions were considered for sending the cancellation notification to a thread.
32806 One would be to define a new SIGCANCEL signal that had the cancellation semantics when
32807 delivered; the other was to define the new *pthread_cancel()* function, which would trigger the
32808 cancellation semantics.

32809 The advantage of a new signal was that so much of the delivery criteria were identical to that
32810 used when trying to deliver a signal that making cancellation notification a signal was seen as
32811 consistent. Indeed, many implementations implement cancellation using a special signal. On the
32812 other hand, there would be no signal functions that could be used with this signal except
32813 *pthread_kill()*, and the behavior of the delivered cancellation signal would be unlike any
32814 previously existing defined signal.

32815 The benefits of a special function include the recognition that this signal would be defined
32816 because of the similar delivery criteria and that this is the only common behavior between a
32817 cancellation request and a signal. In addition, the cancellation delivery mechanism does not
32818 have to be implemented as a signal. There are also strong, if not stronger, parallels with
32819 language exception mechanisms than with signals that are potentially obscured if the delivery
32820 mechanism is visibly closer to signals.

32821 In the end, it was considered that as there were so many exceptions to the use of the new signal
32822 with existing signals functions it would be misleading. A special function has resolved this

32823 problem. This function was carefully defined so that an implementation wishing to provide the
32824 cancellation functions on top of signals could do so. The special function also means that
32825 implementations are not obliged to implement cancellation with signals.

32826 **FUTURE DIRECTIONS**

32827 None.

32828 **SEE ALSO**

32829 *pthread_exit()*, *pthread_cond_timedwait()*, *pthread_join()*, *pthread_setcancelstate()*, the Base
32830 Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

32831 **CHANGE HISTORY**

32832 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

32833 **Issue 6**

32834 The *pthread_cancel()* function is marked as part of the Threads option.

32835 NAME

32836 pthread_cleanup_pop, pthread_cleanup_push — establish cancellation handlers

32837 SYNOPSIS

32838 THR #include <pthread.h>

32839 void pthread_cleanup_pop(int execute);
32840 void pthread_cleanup_push(void (*routine)(void*), void *arg);
32841

32842 DESCRIPTION

32843 The *pthread_cleanup_pop()* function shall remove the routine at the top of the calling thread's
32844 cancellation cleanup stack and optionally invoke it (if *execute* is non-zero).32845 The *pthread_cleanup_push()* function shall push the specified cancellation cleanup handler *routine*
32846 onto the calling thread's cancellation cleanup stack. The cancellation cleanup handler shall be
32847 popped from the cancellation cleanup stack and invoked with the argument *arg* when:

- 32848 • The thread exits (that is, calls *pthread_exit()*).
- 32849 • The thread acts upon a cancellation request.
- 32850 • The thread calls *pthread_cleanup_pop()* with a non-zero *execute* argument.

32851 These functions may be implemented as macros. The application shall ensure that they appear
32852 as statements, and in pairs within the same lexical scope (that is, the *pthread_cleanup_push()*
32853 macro may be thought to expand to a token list whose first token is '{' with
32854 *pthread_cleanup_pop()* expanding to a token list whose last token is the corresponding '}').32855 The effect of calling *longjmp()* or *siglongjmp()* is undefined if there have been any calls to
32856 *pthread_cleanup_push()* or *pthread_cleanup_pop()* made without the matching call since the jump
32857 buffer was filled. The effect of calling *longjmp()* or *siglongjmp()* from inside a cancellation
32858 cleanup handler is also undefined unless the jump buffer was also filled in the cancellation
32859 cleanup handler.32860 The effect of the use of **return**, **break**, **continue**, and **goto** to prematurely leave a code block 2
32861 described by a pair of *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions calls is 2
32862 undefined. 2

32863 RETURN VALUE

32864 The *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions shall not return a value.

32865 ERRORS

32866 No errors are defined.

32867 These functions shall not return an error code of [EINTR].

32868 EXAMPLES

32869 The following is an example using thread primitives to implement a cancelable, writers-priority
32870 read-write lock:

```
32871     typedef struct {
32872         pthread_mutex_t lock;
32873         pthread_cond_t rcond,
32874             wcond;
32875         int lock_count; /* < 0 .. Held by writer. */
32876                     /* > 0 .. Held by lock_count readers. */
32877                     /* = 0 .. Held by nobody. */
32878         int waiting_writers; /* Count of waiting writers. */
32879     } rwlock;
```

```
32880     void
32881     waiting_reader_cleanup(void *arg)
32882     {
32883         rwlock *l;
32884
32885         l = (rwlock *) arg;
32886         pthread_mutex_unlock(&l->lock);
32887     }
32888
32889     void
32890     lock_for_read(rwlock *l)
32891     {
32892         pthread_mutex_lock(&l->lock);
32893         pthread_cleanup_push(waiting_reader_cleanup, l);
32894         while ((l->lock_count < 0) && (l->waiting_writers != 0))
32895             pthread_cond_wait(&l->rcond, &l->lock);
32896         l->lock_count++;
32897         /*
32898          * Note the pthread_cleanup_pop executes
32899          * waiting_reader_cleanup.
32900          */
32901         pthread_cleanup_pop(1);
32902     }
32903
32904     void
32905     release_read_lock(rwlock *l)
32906     {
32907         pthread_mutex_lock(&l->lock);
32908         if (--l->lock_count == 0)
32909             pthread_cond_signal(&l->wcond);
32910         pthread_mutex_unlock(l);
32911     }
32912
32913     void
32914     waiting_writer_cleanup(void *arg)
32915     {
32916         rwlock *l;
32917
32918         l = (rwlock *) arg;
32919         if ((-l->waiting_writers == 0) && (l->lock_count >= 0)) {
32920             /*
32921              * This only happens if we have been canceled.
32922              */
32923             pthread_cond_broadcast(&l->wcond);
32924         }
32925         pthread_mutex_unlock(&l->lock);
32926     }
32927
32928     void
32929     lock_for_write(rwlock *l)
32930     {
32931         pthread_mutex_lock(&l->lock);
32932         l->waiting_writers++;
32933         pthread_cleanup_push(waiting_writer_cleanup, l);
32934         while (l->lock_count != 0)
```

```
32929         pthread_cond_wait(&l->wcond, &l->lock);
32930         l->lock_count = -1;
32931         /*
32932          * Note the pthread_cleanup_pop executes
32933          * waiting_writer_cleanups.
32934          */
32935         pthread_cleanup_pop(1);
32936     }
32937
32938     void
32939     release_write_lock(rwlock *l)
32940     {
32941         pthread_mutex_lock(&l->lock);
32942         l->lock_count = 0;
32943         if (l->waiting_writers == 0)
32944             pthread_cond_broadcast(&l->rcond)
32945         else
32946             pthread_cond_signal(&l->wcond);
32947         pthread_mutex_unlock(&l->lock);
32948     }
32949     /*
32950      * This function is called to initialize the read/write lock.
32951      */
32952     void
32953     initialize_rwlock(rwlock *l)
32954     {
32955         pthread_mutex_init(&l->lock, pthread_mutexattr_default);
32956         pthread_cond_init(&l->wcond, pthread_condattr_default);
32957         pthread_cond_init(&l->rcond, pthread_condattr_default);
32958         l->lock_count = 0;
32959         l->waiting_writers = 0;
32960     }
32961     reader_thread()
32962     {
32963         lock_for_read(&lock);
32964         pthread_cleanup_push(release_read_lock, &lock);
32965         /*
32966          * Thread has read lock.
32967          */
32968         pthread_cleanup_pop(1);
32969     }
32970     writer_thread()
32971     {
32972         lock_for_write(&lock);
32973         pthread_cleanup_push(release_write_lock, &lock);
32974         /*
32975          * Thread has write lock.
32976          */
32977         pthread_cleanup_pop(1);
32978     }
```

32978 APPLICATION USAGE

32979 The two routines that push and pop cancellation cleanup handlers, *pthread_cleanup_push()* and
 32980 *pthread_cleanup_pop()*, can be thought of as left and right parentheses. They always need to be
 32981 matched.

32982 RATIONALE

32983 The restriction that the two routines that push and pop cancellation cleanup handlers,
 32984 *pthread_cleanup_push()* and *pthread_cleanup_pop()*, have to appear in the same lexical scope
 32985 allows for efficient macro or compiler implementations and efficient storage management. A
 32986 sample implementation of these routines as macros might look like this:

```
32987 #define pthread_cleanup_push(rtn,arg) { \
32988     struct _pthread_handler_rec __cleanup_handler, **__head; \
32989     __cleanup_handler.rtn = rtn; \
32990     __cleanup_handler.arg = arg; \
32991     (void) pthread_getspecific(_pthread_handler_key, &__head); \
32992     __cleanup_handler.next = *__head; \
32993     *__head = &__cleanup_handler; \
32994 \
32995 #define pthread_cleanup_pop(ex) \
32996     *__head = __cleanup_handler.next; \
32997     if (ex) (*__cleanup_handler.rtn)(__cleanup_handler.arg); \
32998 }
```

32998 A more ambitious implementation of these routines might do even better by allowing the
 32999 compiler to note that the cancellation cleanup handler is a constant and can be expanded inline.

33000 This volume of IEEE Std 1003.1-2001 currently leaves unspecified the effect of calling *longjmp()*
 33001 from a signal handler executing in a POSIX System Interfaces function. If an implementation
 33002 wants to allow this and give the programmer reasonable behavior, the *longjmp()* function has to
 33003 call all cancellation cleanup handlers that have been pushed but not popped since the time
 33004 *setjmp()* was called.

33005 Consider a multi-threaded function called by a thread that uses signals. If a signal were
 33006 delivered to a signal handler during the operation of *qsort()* and that handler were to call
 33007 *longjmp()* (which, in turn, did *not* call the cancellation cleanup handlers) the helper threads
 33008 created by the *qsort()* function would not be canceled. Instead, they would continue to execute
 33009 and write into the argument array even though the array might have been popped off the stack.

33010 Note that the specified cleanup handling mechanism is especially tied to the C language and,
 33011 while the requirement for a uniform mechanism for expressing cleanup is language-
 33012 independent, the mechanism used in other languages may be quite different. In addition, this
 33013 mechanism is really only necessary due to the lack of a real exception mechanism in the C
 33014 language, which would be the ideal solution.

33015 There is no notion of a cancellation cleanup-safe function. If an application has no cancellation
 33016 points in its signal handlers, blocks any signal whose handler may have cancellation points
 33017 while calling async-unsafe functions, or disables cancellation while calling async-unsafe
 33018 functions, all functions may be safely called from cancellation cleanup routines.

33019 FUTURE DIRECTIONS

33020 None.

33021 SEE ALSO

33022 *pthread_cancel()*, *pthread_setcancelstate()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 33023 <**pthread.h**>

33024 CHANGE HISTORY

33025 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33026 Issue 6

33027 The *pthread_cleanup_pop()* and *pthread_cleanup_push()* functions are marked as part of the
33028 Threads option.

33029 The APPLICATION USAGE section is added.

33030 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

33031 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/88 is applied, updating the 2
33032 DESCRIPTION to describe the consequences of prematurely leaving a code block defined by the 2
33033 *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions. 2

33034 NAME

33035 `pthread_cond_broadcast`, `pthread_cond_signal` — broadcast or signal a condition

33036 SYNOPSIS

33037 THR `#include <pthread.h>`

33038 `int pthread_cond_broadcast(pthread_cond_t *cond);`

33039 `int pthread_cond_signal(pthread_cond_t *cond);`

33040

33041 DESCRIPTION

33042 These functions shall unblock threads blocked on a condition variable.

33043 The `pthread_cond_broadcast()` function shall unblock all threads currently blocked on the
33044 specified condition variable `cond`.

33045 The `pthread_cond_signal()` function shall unblock at least one of the threads that are blocked on
33046 the specified condition variable `cond` (if any threads are blocked on `cond`).

33047 If more than one thread is blocked on a condition variable, the scheduling policy shall determine
33048 the order in which threads are unblocked. When each thread unblocked as a result of a
33049 `pthread_cond_broadcast()` or `pthread_cond_signal()` returns from its call to `pthread_cond_wait()` or
33050 `pthread_cond_timedwait()`, the thread shall own the mutex with which it called
33051 `pthread_cond_wait()` or `pthread_cond_timedwait()`. The thread(s) that are unblocked shall contend
33052 for the mutex according to the scheduling policy (if applicable), and as if each had called
33053 `pthread_mutex_lock()`.

33054 The `pthread_cond_broadcast()` or `pthread_cond_signal()` functions may be called by a thread
33055 whether or not it currently owns the mutex that threads calling `pthread_cond_wait()` or
33056 `pthread_cond_timedwait()` have associated with the condition variable during their waits;
33057 however, if predictable scheduling behavior is required, then that mutex shall be locked by the
33058 thread calling `pthread_cond_broadcast()` or `pthread_cond_signal()`.

33059 The `pthread_cond_broadcast()` and `pthread_cond_signal()` functions shall have no effect if there are
33060 no threads currently blocked on `cond`.

33061 RETURN VALUE

33062 If successful, the `pthread_cond_broadcast()` and `pthread_cond_signal()` functions shall return zero;
33063 otherwise, an error number shall be returned to indicate the error.

33064 ERRORS

33065 The `pthread_cond_broadcast()` and `pthread_cond_signal()` function may fail if:

33066 [EINVAL] The value `cond` does not refer to an initialized condition variable.

33067 These functions shall not return an error code of [EINTR].

33068 EXAMPLES

33069 None.

33070 APPLICATION USAGE

33071 The `pthread_cond_broadcast()` function is used whenever the shared-variable state has been
33072 changed in a way that more than one thread can proceed with its task. Consider a single
33073 producer/multiple consumer problem, where the producer can insert multiple items on a list
33074 that is accessed one item at a time by the consumers. By calling the `pthread_cond_broadcast()`
33075 function, the producer would notify all consumers that might be waiting, and thereby the
33076 application would receive more throughput on a multi-processor. In addition,
33077 `pthread_cond_broadcast()` makes it easier to implement a read-write lock. The
33078 `pthread_cond_broadcast()` function is needed in order to wake up all waiting readers when a

33079 writer releases its lock. Finally, the two-phase commit algorithm can use this broadcast function
 33080 to notify all clients of an impending transaction commit.

33081 It is not safe to use the *pthread_cond_signal()* function in a signal handler that is invoked
 33082 asynchronously. Even if it were safe, there would still be a race between the test of the Boolean
 33083 *pthread_cond_wait()* that could not be efficiently eliminated.

33084 Mutexes and condition variables are thus not suitable for releasing a waiting thread by signaling
 33085 from code running in a signal handler.

33086 RATIONALE

33087 Multiple Awakenings by Condition Signal

33088 On a multi-processor, it may be impossible for an implementation of *pthread_cond_signal()* to
 33089 avoid the unblocking of more than one thread blocked on a condition variable. For example,
 33090 consider the following partial implementation of *pthread_cond_wait()* and *pthread_cond_signal()*,
 33091 executed by two threads in the order given. One thread is trying to wait on the condition
 33092 variable, another is concurrently executing *pthread_cond_signal()*, while a third thread is already
 33093 waiting.

```
33094 pthread_cond_wait(mutex, cond):
33095     value = cond->value; /* 1 */
33096     pthread_mutex_unlock(mutex); /* 2 */
33097     pthread_mutex_lock(cond->mutex); /* 10 */
33098     if (value == cond->value) { /* 11 */
33099         me->next_cond = cond->waiter;
33100         cond->waiter = me;
33101         pthread_mutex_unlock(cond->mutex);
33102         unable_to_run(me);
33103     } else
33104         pthread_mutex_unlock(cond->mutex); /* 12 */
33105     pthread_mutex_lock(mutex); /* 13 */

33106 pthread_cond_signal(cond):
33107     pthread_mutex_lock(cond->mutex); /* 3 */
33108     cond->value++; /* 4 */
33109     if (cond->waiter) { /* 5 */
33110         sleeper = cond->waiter; /* 6 */
33111         cond->waiter = sleeper->next_cond; /* 7 */
33112         able_to_run(sleeper); /* 8 */
33113     }
33114     pthread_mutex_unlock(cond->mutex); /* 9 */
```

33115 The effect is that more than one thread can return from its call to *pthread_cond_wait()* or
 33116 *pthread_cond_timedwait()* as a result of one call to *pthread_cond_signal()*. This effect is called
 33117 “spurious wakeup”. Note that the situation is self-correcting in that the number of threads that
 33118 are so awakened is finite; for example, the next thread to call *pthread_cond_wait()* after the
 33119 sequence of events above blocks.

33120 While this problem could be resolved, the loss of efficiency for a fringe condition that occurs
 33121 only rarely is unacceptable, especially given that one has to check the predicate associated with a
 33122 condition variable anyway. Correcting this problem would unnecessarily reduce the degree of
 33123 concurrency in this basic building block for all higher-level synchronization operations.

33124 An added benefit of allowing spurious wakeups is that applications are forced to code a
 33125 predicate-testing-loop around the condition wait. This also makes the application tolerate

33126 superfluous condition broadcasts or signals on the same condition variable that may be coded in
33127 some other part of the application. The resulting applications are thus more robust. Therefore,
33128 IEEE Std 1003.1-2001 explicitly documents that spurious wakeups may occur.

33129 **FUTURE DIRECTIONS**

33130 None.

33131 **SEE ALSO**

33132 *pthread_cond_destroy()*, *pthread_cond_timedwait()*, the Base Definitions volume of
33133 IEEE Std 1003.1-2001, <**pthread.h**>

33134 **CHANGE HISTORY**

33135 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33136 **Issue 6**

33137 The *pthread_cond_broadcast()* and *pthread_cond_signal()* functions are marked as part of the
33138 Threads option.

33139 The APPLICATION USAGE section is added.

33140 NAME

33141 pthread_cond_destroy, pthread_cond_init — destroy and initialize condition variables

33142 SYNOPSIS

33143 THR #include <pthread.h>

```
33144 int pthread_cond_destroy(pthread_cond_t *cond);
33145 int pthread_cond_init(pthread_cond_t *restrict cond,
33146             const pthread_condattr_t *restrict attr);
33147 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
33148
```

33149 DESCRIPTION

33150 The *pthread_cond_destroy()* function shall destroy the given condition variable specified by *cond*; the object becomes, in effect, uninitialized. An implementation may cause *pthread_cond_destroy()* to set the object referenced by *cond* to an invalid value. A destroyed condition variable object can be reinitialized using *pthread_cond_init()*; the results of otherwise referencing the object after it has been destroyed are undefined.

33155 It shall be safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

33158 The *pthread_cond_init()* function shall initialize the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes shall be used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable shall become initialized.

33162 Only *cond* itself may be used for performing synchronization. The result of referring to copies of *cond* in calls to *pthread_cond_wait()*, *pthread_cond_timedwait()*, *pthread_cond_signal()*, *pthread_cond_broadcast()*, and *pthread_cond_destroy()* is undefined.

33165 Attempting to initialize an already initialized condition variable results in undefined behavior.

33166 In cases where default condition variable attributes are appropriate, the macro PTHREAD_COND_INITIALIZER can be used to initialize condition variables that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to *pthread_cond_init()* with parameter *attr* specified as NULL, except that no error checks are performed.

33170 RETURN VALUE

33171 If successful, the *pthread_cond_destroy()* and *pthread_cond_init()* functions shall return zero; otherwise, an error number shall be returned to indicate the error.

33173 The [EBUSY] and [EINVAL] error checks, if implemented, shall act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the condition variable specified by *cond*.

33176 ERRORS

33177 The *pthread_cond_destroy()* function may fail if:

33178 [EBUSY]	The implementation has detected an attempt to destroy the object referenced by <i>cond</i> while it is referenced (for example, while being used in a <i>pthread_cond_wait()</i> or <i>pthread_cond_timedwait()</i>) by another thread.
----------------------	--

33181 [EINVAL]	The value specified by <i>cond</i> is invalid.
-----------------------	--

33182 The *pthread_cond_init()* function shall fail if:

33183 [EAGAIN]	The system lacked the necessary resources (other than memory) to initialize another condition variable.
-----------------------	---

33185 [ENOMEM] Insufficient memory exists to initialize the condition variable.
 33186 The *pthread_cond_init()* function may fail if:
 33187 [EBUSY] The implementation has detected an attempt to reinitialize the object
 33188 referenced by *cond*, a previously initialized, but not yet destroyed, condition
 33189 variable.
 33190 [EINVAL] The value specified by *attr* is invalid.
 33191 These functions shall not return an error code of [EINTR].

33192 EXAMPLES

33193 A condition variable can be destroyed immediately after all the threads that are blocked on it are
 33194 awakened. For example, consider the following code:

```
33195 struct list {
33196     pthread_mutex_t lm;
33197     ...
33198 }
33199 struct elt {
33200     key k;
33201     int busy;
33202     pthread_cond_t notbusy;
33203     ...
33204 }
33205 /* Find a list element and reserve it. */
33206 struct elt *
33207 list_find(struct list *lp, key k)
33208 {
33209     struct elt *ep;
33210     pthread_mutex_lock(&lp->lm);
33211     while ((ep = find_elt(l, k) != NULL) && ep->busy)
33212         pthread_cond_wait(&ep->notbusy, &lp->lm);
33213     if (ep != NULL)
33214         ep->busy = 1;
33215     pthread_mutex_unlock(&lp->lm);
33216     return(ep);
33217 }
33218 delete_elt(struct list *lp, struct elt *ep)
33219 {
33220     pthread_mutex_lock(&lp->lm);
33221     assert(ep->busy);
33222     ... remove ep from list ...
33223     ep->busy = 0; /* Paranoid. */
33224     (A) pthread_cond_broadcast(&ep->notbusy);
33225     pthread_mutex_unlock(&lp->lm);
33226     (B) pthread_cond_destroy(&ep->notbusy);
33227     free(ep);
33228 }
```

33229 In this example, the condition variable and its list element may be freed (line B) immediately
 33230 after all threads waiting for it are awakened (line A), since the mutex and the code ensure that no
 33231 other thread can touch the element to be deleted.

33232 APPLICATION USAGE

33233 None.

33234 RATIONALE

33235 See *pthread_mutex_init()*; a similar rationale applies to condition variables.

33236 FUTURE DIRECTIONS

33237 None.

33238 SEE ALSO

33239 *pthread_cond_broadcast()*, *pthread_cond_signal()*, *pthread_cond_timedwait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

33241 CHANGE HISTORY

33242 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33243 Issue 6

33244 The *pthread_cond_destroy()* and *pthread_cond_init()* functions are marked as part of the Threads option.

33246 IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

33247 The **restrict** keyword is added to the *pthread_cond_init()* prototype for alignment with the ISO/IEC 9899: 1999 standard.

33249 **NAME**

33250 pthread_cond_signal — signal a condition

33251 **SYNOPSIS**33252 **THR** #include <pthread.h>

33253 int pthread_cond_signal(pthread_cond_t *cond);

33254

33255 **DESCRIPTION**33256 Refer to *pthread_cond_broadcast()*.

33257 NAME

33258 pthread_cond_timedwait, pthread_cond_wait — wait on a condition

33259 SYNOPSIS

33260 THR #include <pthread.h>

```
33261     int pthread_cond_timedwait(pthread_cond_t *restrict cond,
33262         pthread_mutex_t *restrict mutex,
33263         const struct timespec *restrict abstime);
33264     int pthread_cond_wait(pthread_cond_t *restrict cond,
33265         pthread_mutex_t *restrict mutex);
```

33266

33267 DESCRIPTION

33268 The *pthread_cond_timedwait()* and *pthread_cond_wait()* functions shall block on a condition
 33269 variable. They shall be called with *mutex* locked by the calling thread or undefined behavior
 33270 results.

33271 These functions atomically release *mutex* and cause the calling thread to block on the condition
 33272 variable *cond*; atomically here means “atomically with respect to access by another thread to the
 33273 mutex and then the condition variable”. That is, if another thread is able to acquire the mutex
 33274 after the about-to-block thread has released it, then a subsequent call to *pthread_cond_broadcast()*
 33275 or *pthread_cond_signal()* in that thread shall behave as if it were issued after the about-to-block
 33276 thread has blocked.

33277 Upon successful return, the mutex shall have been locked and shall be owned by the calling
 33278 thread.

33279 When using condition variables there is always a Boolean predicate involving shared variables
 33280 associated with each condition wait that is true if the thread should proceed. Spurious wakeups
 33281 from the *pthread_cond_timedwait()* or *pthread_cond_wait()* functions may occur. Since the return
 33282 from *pthread_cond_timedwait()* or *pthread_cond_wait()* does not imply anything about the value
 33283 of this predicate, the predicate should be re-evaluated upon such return.

33284 When a thread waits on a condition variable, having specified a particular mutex to either the 2
 33285 *pthread_cond_timedwait()* or the *pthread_cond_wait()* operation, a dynamic binding is formed 2
 33286 between that mutex and condition variable that remains in effect as long as at least one thread is 2
 33287 blocked on the condition variable. During this time, the effect of an attempt by any thread to 2
 33288 wait on that condition variable using a different mutex is undefined. Once all waiting threads 2
 33289 have been unblocked (as by the *pthread_cond_broadcast()* operation), the next wait operation on 2
 33290 that condition variable shall form a new dynamic binding with the mutex specified by that wait 2
 33291 operation. Even though the dynamic binding between condition variable and mutex may be 2
 33292 removed or replaced between the time a thread is unblocked from a wait on the condition 2
 33293 variable and the time that it returns to the caller or begins cancellation cleanup, the unblocked 2
 33294 thread shall always re-acquire the mutex specified in the condition wait operation call from 2
 33295 which it is returning. 2

33296 A condition wait (whether timed or not) is a cancellation point. When the cancelability type of a 2
 33297 thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a cancellation 2
 33298 request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first 2
 33299 cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute 2
 33300 up to the point of returning from the call to *pthread_cond_timedwait()* or *pthread_cond_wait()*, but 2
 33301 at that point notices the cancellation request and instead of returning to the caller of 2
 33302 *pthread_cond_timedwait()* or *pthread_cond_wait()*, starts the thread cancellation activities, which 2
 33303 includes calling cancellation cleanup handlers. 2

33304 A thread that has been unblocked because it has been canceled while blocked in a call to
33305 *pthread_cond_timedwait()* or *pthread_cond_wait()* shall not consume any condition signal that
33306 may be directed concurrently at the condition variable if there are other threads blocked on the
33307 condition variable.

33308 The *pthread_cond_timedwait()* function shall be equivalent to *pthread_cond_wait()*, except that an
33309 error is returned if the absolute time specified by *abstime* passes (that is, system time equals or
33310 exceeds *abstime*) before the condition *cond* is signaled or broadcasted, or if the absolute time
33311 specified by *abstime* has already been passed at the time of the call.

33312 CS If the Clock Selection option is supported, the condition variable shall have a clock attribute
33313 which specifies the clock that shall be used to measure the time specified by the *abstime*
33314 argument. When such timeouts occur, *pthread_cond_timedwait()* shall nonetheless release and
33315 re-acquire the mutex referenced by *mutex*. The *pthread_cond_timedwait()* function is also a
33316 cancellation point.

33317 If a signal is delivered to a thread waiting for a condition variable, upon return from the signal
33318 handler the thread resumes waiting for the condition variable as if it was not interrupted, or it
33319 shall return zero due to spurious wakeup.

33320 RETURN VALUE

33321 Except in the case of [ETIMEDOUT], all these error checks shall act as if they were performed
33322 immediately at the beginning of processing for the function and shall cause an error return, in
33323 effect, prior to modifying the state of the mutex specified by *mutex* or the condition variable
33324 specified by *cond*.

33325 Upon successful completion, a value of zero shall be returned; otherwise, an error number shall
33326 be returned to indicate the error.

33327 ERRORS

33328 The *pthread_cond_timedwait()* function shall fail if:

33329 [ETIMEDOUT] The time specified by *abstime* to *pthread_cond_timedwait()* has passed. 2

33330 [EINVAL] The value specified by *abstime* is invalid. 2

33331 These functions may fail if: 2

33332 [EINVAL] The value specified by *cond* or *mutex* is invalid. 2

33333 [EPERM] The mutex was not owned by the current thread at the time of the call.

33334 These functions shall not return an error code of [EINTR].

33335 EXAMPLES

33336 None.

33337 APPLICATION USAGE

33338 None.

33339 RATIONALE

33340	Condition Wait Semantics	
33341	It is important to note that when <i>pthread_cond_wait()</i> and <i>pthread_cond_timedwait()</i> return without error, the associated predicate may still be false. Similarly, when <i>pthread_cond_timedwait()</i> returns with the timeout error, the associated predicate may be true due to an unavoidable race between the expiration of the timeout and the predicate state change.	2
33345	The application needs to recheck the predicate on any return because it cannot be sure there is another thread waiting on the thread to handle the signal, and if there is not then the signal is lost. The burden is on the application to check the predicate.	2
33346		2
33347		2
33348	Some implementations, particularly on a multi-processor, may sometimes cause multiple threads to wake up when the condition variable is signaled simultaneously on different processors.	
33349		
33350		
33351	In general, whenever a condition wait returns, the thread has to re-evaluate the predicate associated with the condition wait to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait does not imply that the associated predicate is either true or false.	
33352		
33353		
33354		
33355	It is thus recommended that a condition wait be enclosed in the equivalent of a “while loop” that checks the predicate.	
33356		
33357	Timed Wait Semantics	
33358	An absolute time measure was chosen for specifying the timeout parameter for two reasons.	
33359	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	
33360	of a function that specifies relative timeouts. For example, assume that <i>clock_gettime()</i> returns the current time and <i>cond_relative_timed_wait()</i> uses relative timeouts:	
33361		
33362		
33363	<i>clock_gettime(CLOCK_REALTIME, &now)</i>	
33364	<i>reltime = sleep_til_this_absolute_time -now;</i>	
33365	<i>cond_relative_timed_wait(c, m, &reltime);</i>	
33366	If the thread is preempted between the first statement and the last statement, the thread blocks	
33367	for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout	
33368	also need not be recomputed if it is used multiple times in a loop, such as that enclosing a	
33369	condition wait.	
33370	For cases when the system clock is advanced discontinuously by an operator, it is expected that	
33371	implementations process any timed wait expiring at an intervening time as if that time had	
33372	actually occurred.	
33373	Cancellation and Condition Wait	
33374	A condition wait, whether timed or not, is a cancellation point. That is, the functions	
33375	<i>pthread_cond_wait()</i> or <i>pthread_cond_timedwait()</i> are points where a pending (or concurrent)	
33376	cancellation request is noticed. The reason for this is that an indefinite wait is possible at these	
33377	points—whatever event is being waited for, even if the program is totally correct, might never	
33378	occur; for example, some input data being awaited might never be sent. By making condition	
33379	wait a cancellation point, the thread can be canceled and perform its cancellation cleanup	
33380	handler even though it may be stuck in some indefinite wait.	
33381	A side effect of acting on a cancellation request while a thread is blocked on a condition variable	
33382	is to re-acquire the mutex before calling any of the cancellation cleanup handlers. This is done in	
33383	order to ensure that the cancellation cleanup handler is executed in the same state as the critical	
33384	code that lies both before and after the call to the condition wait function. This rule is also	

33385 required when interfacing to POSIX threads from languages, such as Ada or C++, which may
33386 choose to map cancellation onto a language exception; this rule ensures that each exception
33387 handler guarding a critical section can always safely depend upon the fact that the associated
33388 mutex has already been locked regardless of exactly where within the critical section the
33389 exception was raised. Without this rule, there would not be a uniform rule that exception
33390 handlers could follow regarding the lock, and so coding would become very cumbersome.

33391 Therefore, since *some* statement has to be made regarding the state of the lock when a
33392 cancellation is delivered during a wait, a definition has been chosen that makes application
33393 coding most convenient and error free.

33394 When acting on a cancellation request while a thread is blocked on a condition variable, the
33395 implementation is required to ensure that the thread does not consume any condition signals
33396 directed at that condition variable if there are any other threads waiting on that condition
33397 variable. This rule is specified in order to avoid deadlock conditions that could occur if these two
33398 independent requests (one acting on a thread and the other acting on the condition variable)
33399 were not processed independently.

33400 **Performance of Mutexes and Condition Variables**

33401 Mutexes are expected to be locked only for a few instructions. This practice is almost
33402 automatically enforced by the desire of programmers to avoid long serial regions of execution
33403 (which would reduce total effective parallelism).

33404 When using mutexes and condition variables, one tries to ensure that the usual case is to lock the
33405 mutex, access shared data, and unlock the mutex. Waiting on a condition variable should be a
33406 relatively rare situation. For example, when implementing a read-write lock, code that acquires a
33407 read-lock typically needs only to increment the count of readers (under mutual-exclusion) and
33408 return. The calling thread would actually wait on the condition variable only when there is
33409 already an active writer. So the efficiency of a synchronization operation is bounded by the cost
33410 of mutex lock/unlock and not by condition wait. Note that in the usual case there is no context
33411 switch.

33412 This is not to say that the efficiency of condition waiting is unimportant. Since there needs to be
33413 at least one context switch per Ada rendezvous, the efficiency of waiting on a condition variable
33414 is important. The cost of waiting on a condition variable should be little more than the minimal
33415 cost for a context switch plus the time to unlock and lock the mutex.

33416 **Features of Mutexes and Condition Variables**

33417 It had been suggested that the mutex acquisition and release be decoupled from condition wait.
33418 This was rejected because it is the combined nature of the operation that, in fact, facilitates
33419 realtime implementations. Those implementations can atomically move a high-priority thread
33420 between the condition variable and the mutex in a manner that is transparent to the caller. This
33421 can prevent extra context switches and provide more deterministic acquisition of a mutex when
33422 the waiting thread is signaled. Thus, fairness and priority issues can be dealt with directly by the
33423 scheduling discipline. Furthermore, the current condition wait operation matches existing
33424 practice.

33425 **Scheduling Behavior of Mutexes and Condition Variables**

33426 Synchronization primitives that attempt to interfere with scheduling policy by specifying an
 33427 ordering rule are considered undesirable. Threads waiting on mutexes and condition variables
 33428 are selected to proceed in an order dependent upon the scheduling policy rather than in some
 33429 fixed order (for example, FIFO or priority). Thus, the scheduling policy determines which
 33430 thread(s) are awakened and allowed to proceed.

33431 **Timed Condition Wait**

33432 The *pthread_cond_timedwait()* function allows an application to give up waiting for a particular
 33433 condition after a given amount of time. An example of its use follows:

```
33434       (void) pthread_mutex_lock(&t.mn);
33435               t.waiters++;
33436               clock_gettime(CLOCK_REALTIME, &ts);
33437               ts.tv_sec += 5;
33438               rc = 0;
33439               while (! mypredicate(&t) && rc == 0)
33440                rc = pthread_cond_timedwait(&t.cond, &t.mn, &ts);
33441               t.waiters--;
33442               if (rc == 0) setmystate(&t);
33443       (void) pthread_mutex_unlock(&t.mn);
```

33444 By making the timeout parameter absolute, it does not need to be recomputed each time the
 33445 program checks its blocking predicate. If the timeout was relative, it would have to be
 33446 recomputed before each call. This would be especially difficult since such code would need to
 33447 take into account the possibility of extra wakeups that result from extra broadcasts or signals on
 33448 the condition variable that occur before either the predicate is true or the timeout is due.

33449 **FUTURE DIRECTIONS**

33450 None.

33451 **SEE ALSO**

33452 *pthread_cond_signal()*, *pthread_cond_broadcast()*, the Base Definitions volume of
 33453 IEEE Std 1003.1-2001, <**pthread.h**>

33454 **CHANGE HISTORY**

33455 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33456 **Issue 6**

33457 The *pthread_cond_timedwait()* and *pthread_cond_wait()* functions are marked as part of the
 33458 Threads option.

33459 The Open Group Corrigendum U021/9 is applied, correcting the prototype for the
 33460 *pthread_cond_wait()* function.

33461 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for
 33462 the Clock Selection option.

33463 The ERRORS section has an additional case for [EPERM] in response to IEEE PASC
 33464 Interpretation 1003.1c #28.

33465 The **restrict** keyword is added to the *pthread_cond_timedwait()* and *pthread_cond_wait()*
 33466 prototypes for alignment with the ISO/IEC 9899:1999 standard.

33467 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/89 is applied, updating the 2
 33468 DESCRIPTION for consistency with the *pthread_cond_destroy()* function that states it is safe to 2
 33469 destroy an initialized condition variable upon which no threads are currently blocked. 2

33470	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/90 is applied, updating words in the DESCRIPTION from “the cancelability enable state” to “the cancelability type”.	2
33471		2
33472	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/91 is applied, updating the ERRORS section to remove the error case related to <i>abstime</i> from the <i>pthread_cond_wait()</i> function, and to make the error case related to <i>abstime</i> mandatory for <i>pthread_cond_timedwait()</i> for consistency with other functions.	2
33473		2
33474		2
33475		2
33476	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/92 is applied, adding a new paragraph to the RATIONALE section stating that an application should check the predicate on any return from this function.	2
33477		2
33478		2

33479 NAME

33480 **pthread_condattr_destroy**, **pthread_condattr_init** — destroy and initialize the condition variable
33481 attributes object

33482 SYNOPSIS

33483 THR `#include <pthread.h>`
33484 `int pthread_condattr_destroy(pthread_condattr_t *attr);`
33485 `int pthread_condattr_init(pthread_condattr_t *attr);`
33486

33487 DESCRIPTION

33488 The *pthread_condattr_destroy()* function shall destroy a condition variable attributes object; the
33489 object becomes, in effect, uninitialized. An implementation may cause *pthread_condattr_destroy()*
33490 to set the object referenced by *attr* to an invalid value. A destroyed *attr* attributes object can be
33491 reinitialized using *pthread_condattr_init()*; the results of otherwise referencing the object after it
33492 has been destroyed are undefined.

33493 The *pthread_condattr_init()* function shall initialize a condition variable attributes object *attr* with
33494 the default value for all of the attributes defined by the implementation.

33495 Results are undefined if *pthread_condattr_init()* is called specifying an already initialized *attr*
33496 attributes object.

33497 After a condition variable attributes object has been used to initialize one or more condition
33498 variables, any function affecting the attributes object (including destruction) shall not affect any
33499 previously initialized condition variables.

33500 This volume of IEEE Std 1003.1-2001 requires two attributes, the *clock* attribute and the *process-*
33501 *shared* attribute.

33502 Additional attributes, their default values, and the names of the associated functions to get and
33503 set those attribute values are implementation-defined.

33504 RETURN VALUE

33505 If successful, the *pthread_condattr_destroy()* and *pthread_condattr_init()* functions shall return
33506 zero; otherwise, an error number shall be returned to indicate the error.

33507 ERRORS

33508 The *pthread_condattr_destroy()* function may fail if:

33509 [EINVAL] The value specified by *attr* is invalid.

33510 The *pthread_condattr_init()* function shall fail if:

33511 [ENOMEM] Insufficient memory exists to initialize the condition variable attributes object.

33512 These functions shall not return an error code of [EINTR].

33513 EXAMPLES

33514 None.

33515 APPLICATION USAGE

33516 None.

33517 RATIONALE

33518 See *pthread_attr_init()* and *pthread_mutex_init()*.

33519 A *process-shared* attribute has been defined for condition variables for the same reason it has been
33520 defined for mutexes.

33521 FUTURE DIRECTIONS

33522 None.

33523 SEE ALSO

33524 *pthread_attr_destroy()*, *pthread_cond_destroy()*, *pthread_condattr_getpshared()*, *pthread_create()*,
33525 *pthread_mutex_destroy()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

33526 CHANGE HISTORY

33527 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33528 Issue 6

33529 The *pthread_condattr_destroy()* and *pthread_condattr_init()* functions are marked as part of the
33530 Threads option.

33531 NAME

33532 *pthread_condattr_getclock*, *pthread_condattr_setclock* — get and set the clock selection
 33533 condition variable attribute (**ADVANCED REALTIME**)

33534 SYNOPSIS

```
33535 THR CS #include <pthread.h>
33536
33537     int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
33538             clockid_t *restrict clock_id);
33539     int pthread_condattr_setclock(pthread_condattr_t *attr,
33540             clockid_t clock_id);
```

33541 DESCRIPTION

33542 The *pthread_condattr_getclock()* function shall obtain the value of the *clock* attribute from the
 33543 attributes object referenced by *attr*. The *pthread_condattr_setclock()* function shall set the *clock*
 33544 attribute in an initialized attributes object referenced by *attr*. If *pthread_condattr_setclock()* is
 33545 called with a *clock_id* argument that refers to a CPU-time clock, the call shall fail.

33546 The *clock* attribute is the clock ID of the clock that shall be used to measure the timeout service of
 33547 *pthread_cond_timedwait()*. The default value of the *clock* attribute shall refer to the system clock.

33548 RETURN VALUE

33549 If successful, the *pthread_condattr_getclock()* function shall return zero and store the value of the
 33550 clock attribute of *attr* into the object referenced by the *clock_id* argument. Otherwise, an error
 33551 number shall be returned to indicate the error.

33552 If successful, the *pthread_condattr_setclock()* function shall return zero; otherwise, an error
 33553 number shall be returned to indicate the error.

33554 ERRORS

33555 These functions may fail if:

33556 [EINVAL] The value specified by *attr* is invalid.

33557 The *pthread_condattr_setclock()* function may fail if:

33558 [EINVAL] The value specified by *clock_id* does not refer to a known clock, or is a CPU-
 33559 time clock.

33560 These functions shall not return an error code of [EINTR].

33561 EXAMPLES

33562 None.

33563 APPLICATION USAGE

33564 None.

33565 RATIONALE

33566 None.

33567 FUTURE DIRECTIONS

33568 None.

33569 SEE ALSO

33570 *pthread_cond_destroy()*, *pthread_cond_timedwait()*, *pthread_condattr_destroy()*,
 33571 *pthread_condattr_getpshared()* (on page 1060),¹ *pthread_condattr_init()*,
 33572 *pthread_condattr_setpshared()* (on page 1064),¹ *pthread_create()*, *pthread_mutex_init()*, the Base
 33573 Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

33574 CHANGE HISTORY

33575 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

33576 NAME

33577 pthread_condattr_getpshared, pthread_condattr_setpshared — get and set the process-shared
33578 condition variable attributes

33579 SYNOPSIS

33580 THR TSH #include <pthread.h>

```
33581     int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,  
33582             int *restrict pshared);  
33583     int pthread_condattr_setpshared(pthread_condattr_t *attr,  
33584             int pshared);  
33585
```

33586 DESCRIPTION

33587 The *pthread_condattr_getpshared()* function shall obtain the value of the *process-shared* attribute
33588 from the attributes object referenced by *attr*. The *pthread_condattr_setpshared()* function shall set
33589 the *process-shared* attribute in an initialized attributes object referenced by *attr*.

33590 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a condition
33591 variable to be operated upon by any thread that has access to the memory where the condition
33592 variable is allocated, even if the condition variable is allocated in memory that is shared by
33593 multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the
33594 condition variable shall only be operated upon by threads created within the same process as the
33595 thread that initialized the condition variable; if threads of differing processes attempt to operate
33596 on such a condition variable, the behavior is undefined. The default value of the attribute is
33597 PTHREAD_PROCESS_PRIVATE.

33598 RETURN VALUE

33599 If successful, the *pthread_condattr_setpshared()* function shall return zero; otherwise, an error
33600 number shall be returned to indicate the error.

33601 If successful, the *pthread_condattr_getpshared()* function shall return zero and store the value of
33602 the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise,
33603 an error number shall be returned to indicate the error.

33604 ERRORS

33605 The *pthread_condattr_getpshared()* and *pthread_condattr_setpshared()* functions may fail if:
33606 [EINVAL] The value specified by *attr* is invalid.
33607 The *pthread_condattr_setpshared()* function may fail if:
33608 [EINVAL] The new value specified for the attribute is outside the range of legal values
33609 for that attribute.
33610 These functions shall not return an error code of [EINTR].

33611 EXAMPLES

33612 None.

33613 APPLICATION USAGE

33614 None.

33615 RATIONALE

33616 None.

33617 FUTURE DIRECTIONS

33618 None.

33619 SEE ALSO

33620 *pthread_create()*, *pthread_cond_destroy()*, *pthread_condattr_destroy()*, *pthread_mutex_destroy()*, the
33621 Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

33622 CHANGE HISTORY

33623 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33624 Issue 6

33625 The *pthread_condattr_getpshared()* and *pthread_condattr_setpshared()* functions are marked as part
33626 of the Threads and Thread Process-Shared Synchronization options.

33627 The **restrict** keyword is added to the *pthread_condattr_getpshared()* prototype for alignment with
33628 the ISO/IEC 9899:1999 standard.

33629 **NAME**33630 `pthread_condattr_init` — initialize the condition variable attributes object33631 **SYNOPSIS**33632 `THR #include <pthread.h>`33633 `int pthread_condattr_init(pthread_condattr_t *attr);`

33634

33635 **DESCRIPTION**33636 Refer to *pthread_condattr_destroy()*.

33637 NAME

33638 `pthread_condattr_setclock` — set the clock selection condition variable attribute

33639 SYNOPSIS

33640 THR CS `#include <pthread.h>`

```
33641        int pthread_condattr_setclock(pthread_condattr_t *attr,
33642              clockid_t clock_id);
```

33643

33644 DESCRIPTION

33645 Refer to `pthread_condattr_getclock()`.

33646 NAME

33647 pthread_condattr_setpshared — set the process-shared condition variable attribute

33648 SYNOPSIS

33649 THR TSH #include <pthread.h>

```
33650     int pthread_condattr_setpshared(pthread_condattr_t *attr,  
33651         int pshared);  
33652
```

33653 DESCRIPTION

33654 Refer to *pthread_condattr_getpshared()*.

33655 NAME

33656 pthread_create — thread creation

33657 SYNOPSIS

```
33658 THR #include <pthread.h>
33659     int pthread_create(pthread_t *restrict thread,
33660             const pthread_attr_t *restrict attr,
33661             void *(*start_routine)(void*), void *restrict arg);
```

33663 DESCRIPTION

33664 The *pthread_create()* function shall create a new thread, with attributes specified by *attr*, within a
 33665 process. If *attr* is NULL, the default attributes shall be used. If the attributes specified by *attr* are
 33666 modified later, the thread's attributes shall not be affected. Upon successful completion,
 33667 *pthread_create()* shall store the ID of the created thread in the location referenced by *thread*.

33668 The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine*
 33669 returns, the effect shall be as if there was an implicit call to *pthread_exit()* using the return value
 33670 of *start_routine* as the exit status. Note that the thread in which *main()* was originally invoked
 33671 differs from this. When it returns from *main()*, the effect shall be as if there was an implicit call
 33672 to *exit()* using the return value of *main()* as the exit status.

33673 The signal state of the new thread shall be initialized as follows:

- The signal mask shall be inherited from the creating thread.
- The set of signals pending for the new thread shall be empty.

33676 XSI The alternate stack shall not be inherited. 1

33677 The floating-point environment shall be inherited from the creating thread.

33678 If *pthread_create()* fails, no new thread is created and the contents of the location referenced by
 33679 *thread* are undefined.

33680 TCT If *_POSIX_THREAD_CPUTIME* is defined, the new thread shall have a CPU-time clock
 33681 accessible, and the initial value of this clock shall be set to zero.

33682 RETURN VALUE

33683 If successful, the *pthread_create()* function shall return zero; otherwise, an error number shall be
 33684 returned to indicate the error.

33685 ERRORS

33686 The *pthread_create()* function shall fail if:

33687 [EAGAIN] The system lacked the necessary resources to create another thread, or the
 33688 system-imposed limit on the total number of threads in a process
 33689 {PTHREAD_THREADS_MAX} would be exceeded. 2

33690 [EPERM] The caller does not have appropriate permission to set the required
 33691 scheduling parameters or scheduling policy.

33692 The *pthread_create()* function may fail if: 2

33693 [EINVAL] The attributes specified by *attr* are invalid. 2

33694 The *pthread_create()* function shall not return an error code of [EINTR].

33695 **EXAMPLES**

33696 None.

33697 **APPLICATION USAGE**

33698 There is no requirement on the implementation that the ID of the created thread be available 2
33699 before the newly created thread starts executing. The calling thread can obtain the ID of the 2
33700 created thread through the return value of the *pthread_create()* function, and the newly created 2
33701 thread can obtain its ID by a call to *pthread_self()*. 2

33702 **RATIONALE**

33703 A suggested alternative to *pthread_create()* would be to define two separate operations: create 2
33704 and start. Some applications would find such behavior more natural. Ada, in particular, 2
33705 separates the “creation” of a task from its “activation”. 2

33706 Splitting the operation was rejected by the standard developers for many reasons:

- 33707 • The number of calls required to start a thread would increase from one to two and thus place 2
33708 an additional burden on applications that do not require the additional synchronization. The 2
33709 second call, however, could be avoided by the additional complication of a start-up state 2
33710 attribute.
- 33711 • An extra state would be introduced: “created but not started”. This would require the 2
33712 standard to specify the behavior of the thread operations when the target has not yet started 2
33713 executing.
- 33714 • For those applications that require such behavior, it is possible to simulate the two separate 2
33715 steps with the facilities that are currently provided. The *start_routine()* can synchronize by 2
33716 waiting on a condition variable that is signaled by the start operation.

33717 An Ada implementor can choose to create the thread at either of two points in the Ada program: 2
33718 when the task object is created, or when the task is activated (generally at a “begin”). If the first 2
33719 approach is adopted, the *start_routine()* needs to wait on a condition variable to receive the 2
33720 order to begin “activation”. The second approach requires no such condition variable or extra 2
33721 synchronization. In either approach, a separate Ada task control block would need to be created 2
33722 when the task object is created to hold rendezvous queues, and so on.

33723 An extension of the preceding model would be to allow the state of the thread to be modified 2
33724 between the create and start. This would allow the thread attributes object to be eliminated. This 2
33725 has been rejected because:

- 33726 • All state in the thread attributes object has to be able to be set for the thread. This would 2
33727 require the definition of functions to modify thread attributes. There would be no reduction 2
33728 in the number of function calls required to set up the thread. In fact, for an application that 2
33729 creates all threads using identical attributes, the number of function calls required to set up 2
33730 the threads would be dramatically increased. Use of a thread attributes object permits the 2
33731 application to make one set of attribute setting function calls. Otherwise, the set of attribute 2
33732 setting function calls needs to be made for each thread creation.
- 33733 • Depending on the implementation architecture, functions to set thread state would require 2
33734 kernel calls, or for other implementation reasons would not be able to be implemented as 2
33735 macros, thereby increasing the cost of thread creation.
- 33736 • The ability for applications to segregate threads by class would be lost.

33737 Another suggested alternative uses a model similar to that for process creation, such as “thread 2
33738 fork”. The fork semantics would provide more flexibility and the “create” function can be 2
33739 implemented simply by doing a thread fork followed immediately by a call to the desired “start 2
33740 routine” for the thread. This alternative has these problems:

- For many implementations, the entire stack of the calling thread would need to be duplicated, since in many architectures there is no way to determine the size of the calling frame.
 - Efficiency is reduced since at least some part of the stack has to be copied, even though in most cases the thread never needs the copied context, since it merely calls the desired start routine.

33747 FUTURE DIRECTIONS

33748 None.

33749 SEE ALSO

33750 *fork(), pthread_exit(), pthread_join()*, the Base Definitions volume of IEEE Std 1003.1-2001,
33751 <**pthread.h**>

33752 CHANGE HISTORY

33753 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33754 Issue 6

33755 The *pthread_create()* function is marked as part of the Threads option.

33756 The following new requirements on POSIX implementations derive from alignment with the
33757 Single UNIX Specification:

- The [EPERM] mandatory error condition is added.

33759 The thread CPU-time clock semantics are added for alignment with IEEE Std 1003.1d-1999.

33760 The **restrict** keyword is added to the *pthread_create()* prototype for alignment with the
33761 ISO/IEC 9899: 1999 standard.

33762 The DESCRIPTION is updated to make it explicit that the floating-point environment is
33763 inherited from the creating thread.

33764 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/44 is applied, adding text that the
33765 alternate stack is not inherited.

33766 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/93 is applied, updating the ERRORS
33767 section to remove the mandatory [EINVAL] error (“The value specified by *attr* is invalid”), and
33768 adding the optional [EINVAL] error (“The attributes specified by *attr* are invalid”).

33769 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/94 is applied, adding the APPLICATION
33770 USAGE section.

33771 NAME

33772 `pthread_detach` — detach a thread

33773 SYNOPSIS

33774 THR

```
#include <pthread.h>
```

33775

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

33776

33777 DESCRIPTION

33778 The `pthread_detach()` function shall indicate to the implementation that storage for the thread
33779 *thread* can be reclaimed when that thread terminates. If *thread* has not terminated,
3380 `pthread_detach()` shall not cause it to terminate. The effect of multiple `pthread_detach()` calls on
3381 the same target thread is unspecified.

33782 RETURN VALUE

33783 If the call succeeds, `pthread_detach()` shall return 0; otherwise, an error number shall be returned
33784 to indicate the error.

33785 ERRORS

33786 The `pthread_detach()` function may fail if:

2

33787 [EINVAL] The implementation has detected that the value specified by *thread* does not
33788 refer to a joinable thread.

33789 [ESRCH] No thread could be found corresponding to that specified by the given thread
33790 ID.

33791 The `pthread_detach()` function shall not return an error code of [EINTR].

33792 EXAMPLES

33793 None.

33794 APPLICATION USAGE

33795 None.

33796 RATIONALE

33797 The `pthread_join()` or `pthread_detach()` functions should eventually be called for every thread that
33798 is created so that storage associated with the thread may be reclaimed.

33799 It has been suggested that a “detach” function is not necessary; the *detachstate* thread creation
33800 attribute is sufficient, since a thread need never be dynamically detached. However, need arises
33801 in at least two cases:

- 33802 1. In a cancellation handler for a `pthread_join()` it is nearly essential to have a `pthread_detach()`
33803 function in order to detach the thread on which `pthread_join()` was waiting. Without it, it
33804 would be necessary to have the handler do another `pthread_join()` to attempt to detach the
33805 thread, which would both delay the cancellation processing for an unbounded period and
33806 introduce a new call to `pthread_join()`, which might itself need a cancellation handler. A
33807 dynamic detach is nearly essential in this case.
- 33808 2. In order to detach the “initial thread” (as may be desirable in processes that set up server
33809 threads).

33810 FUTURE DIRECTIONS

33811 None.

33812 SEE ALSO

33813 *pthread_join()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

33814 CHANGE HISTORY

33815 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33816 Issue 6

33817 The *pthread_detach()* function is marked as part of the Threads option.

33818 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/95 is applied, updating the ERRORS 2
33819 section so that the [EINVAL] and [ESRCH] error cases become optional. 2

33820 NAME

33821 *pthread_equal* — compare thread IDs

33822 SYNOPSIS

33823 **THR** `#include <pthread.h>`

33824 `int pthread_equal(pthread_t t1, pthread_t t2);`

33825

33826 DESCRIPTION

33827 This function shall compare the thread IDs *t1* and *t2*.

33828 RETURN VALUE

33829 The *pthread_equal()* function shall return a non-zero value if *t1* and *t2* are equal; otherwise, zero shall be returned.

33831 If either *t1* or *t2* are not valid thread IDs, the behavior is undefined.

33832 ERRORS

33833 No errors are defined.

33834 The *pthread_equal()* function shall not return an error code of [EINTR].

33835 EXAMPLES

33836 None.

33837 APPLICATION USAGE

33838 None.

33839 RATIONALE

33840 Implementations may choose to define a thread ID as a structure. This allows additional flexibility and robustness over using an **int**. For example, a thread ID could include a sequence number that allows detection of “dangling IDs” (copies of a thread ID that has been detached). Since the C language does not support comparison on structure types, the *pthread_equal()* function is provided to compare thread IDs.

33845 FUTURE DIRECTIONS

33846 None.

33847 SEE ALSO

33848 *pthread_create()*, *pthread_self()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

33849 CHANGE HISTORY

33850 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33851 Issue 6

33852 The *pthread_equal()* function is marked as part of the Threads option.

33853 **NAME**

33854 pthread_exit — thread termination

33855 **SYNOPSIS**

33856 THR `#include <pthread.h>`

33857 `void pthread_exit(void *value_ptr);`

33858

33859 **DESCRIPTION**

33860 The *pthread_exit()* function shall terminate the calling thread and make the value *value_ptr* available to any successful join with the terminating thread. Any cancellation cleanup handlers that have been pushed and not yet popped shall be popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions shall be called in an unspecified order. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process-level cleanup actions, including, but not limited to, calling any *atexit()* routines that may exist.

33868 An implicit call to *pthread_exit()* is made when a thread other than the thread in which *main()* was first invoked returns from the start routine that was used to create it. The function's return value shall serve as the thread's exit status.

33871 The behavior of *pthread_exit()* is undefined if called from a cancellation cleanup handler or 33872 destructor function that was invoked as a result of either an implicit or explicit call to 33873 *pthread_exit()*.

33874 After a thread has terminated, the result of access to local (auto) variables of the thread is 33875 undefined. Thus, references to local variables of the exiting thread should not be used for the 33876 *pthread_exit()* *value_ptr* parameter value.

33877 The process shall exit with an exit status of 0 after the last thread has been terminated. The 33878 behavior shall be as if the implementation called *exit()* with a zero argument at thread 33879 termination time.

33880 **RETURN VALUE**

33881 The *pthread_exit()* function cannot return to its caller.

33882 **ERRORS**

33883 No errors are defined.

33884 **EXAMPLES**

33885 None.

33886 **APPLICATION USAGE**

33887 None.

33888 **RATIONALE**

33889 The normal mechanism by which a thread terminates is to return from the routine that was 33890 specified in the *pthread_create()* call that started it. The *pthread_exit()* function provides the 33891 capability for a thread to terminate without requiring a return from the start routine of that 33892 thread, thereby providing a function analogous to *exit()*.

33893 Regardless of the method of thread termination, any cancellation cleanup handlers that have 33894 been pushed and not yet popped are executed, and the destructors for any existing thread- 33895 specific data are executed. This volume of IEEE Std 1003.1-2001 requires that cancellation 33896 cleanup handlers be popped and called in order. After all cancellation cleanup handlers have 33897 been executed, thread-specific data destructors are called, in an unspecified order, for each item 33898 of thread-specific data that exists in the thread. This ordering is necessary because cancellation

33899 cleanup handlers may rely on thread-specific data.

33900 As the meaning of the status is determined by the application (except when the thread has been
33901 canceled, in which case it is PTHREAD_CANCELED), the implementation has no idea what an
33902 illegal status value is, which is why no address error checking is done.

33903 FUTURE DIRECTIONS

33904 None.

33905 SEE ALSO

33906 *exit()*, *pthread_create()*, *pthread_join()*, the Base Definitions volume of IEEE Std 1003.1-2001,
33907 <pthread.h>

33908 CHANGE HISTORY

33909 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

33910 Issue 6

33911 The *pthread_exit()* function is marked as part of the Threads option.

33912 NAME

33913 pthread_getconcurrency, pthread_setconcurrency — get and set the level of concurrency

33914 SYNOPSIS

33915 XSI #include <pthread.h>

```
33916        int pthread_getconcurrency(void);  
33917        int pthread_setconcurrency(int new_level);
```

33918

33919 DESCRIPTION

33920 Unbound threads in a process may or may not be required to be simultaneously active. By
33921 default, the threads implementation ensures that a sufficient number of threads are active so that
33922 the process can continue to make progress. While this conserves system resources, it may not
33923 produce the most effective level of concurrency.

33924 The *pthread_setconcurrency()* function allows an application to inform the threads
33925 implementation of its desired concurrency level, *new_level*. The actual level of concurrency
33926 provided by the implementation as a result of this function call is unspecified.

33927 If *new_level* is zero, it causes the implementation to maintain the concurrency level at its
33928 discretion as if *pthread_setconcurrency()* had never been called.

33929 The *pthread_getconcurrency()* function shall return the value set by a previous call to the
33930 *pthread_setconcurrency()* function. If the *pthread_setconcurrency()* function was not previously
33931 called, this function shall return zero to indicate that the implementation is maintaining the
33932 concurrency level.

33933 A call to *pthread_setconcurrency()* shall inform the implementation of its desired concurrency
33934 level. The implementation shall use this as a hint, not a requirement.

33935 If an implementation does not support multiplexing of user threads on top of several kernel-
33936 scheduled entities, the *pthread_setconcurrency()* and *pthread_getconcurrency()* functions are
33937 provided for source code compatibility but they shall have no effect when called. To maintain
33938 the function semantics, the *new_level* parameter is saved when *pthread_setconcurrency()* is called
33939 so that a subsequent call to *pthread_getconcurrency()* shall return the same value.

33940 RETURN VALUE

33941 If successful, the *pthread_setconcurrency()* function shall return zero; otherwise, an error number
33942 shall be returned to indicate the error.

33943 The *pthread_getconcurrency()* function shall always return the concurrency level set by a previous
33944 call to *pthread_setconcurrency()*. If the *pthread_setconcurrency()* function has never been called,
33945 *pthread_getconcurrency()* shall return zero.

33946 ERRORS

33947 The *pthread_setconcurrency()* function shall fail if:

33948 [EINVAL] The value specified by *new_level* is negative.

33949 [EAGAIN] The value specified by *new_level* would cause a system resource to be exceeded.

33950 These functions shall not return an error code of [EINTR].

33951 EXAMPLES

33952 None.

33953 APPLICATION USAGE

33954 Use of these functions changes the state of the underlying concurrency upon which the application depends. Library developers are advised to not use the *pthread_getconcurrency()* and *pthread_setconcurrency()* functions since their use may conflict with an applications use of these functions.

33958 RATIONALE

33959 None.

33960 FUTURE DIRECTIONS

33961 None.

33962 SEE ALSO

33963 The Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

33964 CHANGE HISTORY

33965 First released in Issue 5.

33966 NAME

33967 pthread_getcpuclockid — access a thread CPU-time clock (**ADVANCED REALTIME**
33968 **THREADS**)

33969 SYNOPSIS

```
33970 THR TCT #include <pthread.h>
33971         #include <time.h>
33972         int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);
33973
```

33974 DESCRIPTION

33975 The *pthread_getcpuclockid()* function shall return in *clock_id* the clock ID of the CPU-time clock of
33976 the thread specified by *thread_id*, if the thread specified by *thread_id* exists.

33977 RETURN VALUE

33978 Upon successful completion, *pthread_getcpuclockid()* shall return zero; otherwise, an error
33979 number shall be returned to indicate the error.

33980 ERRORS

33981 The *pthread_getcpuclockid()* function may fail if:

33982 [ESRCH] The value specified by *thread_id* does not refer to an existing thread.

33983 EXAMPLES

33984 None.

33985 APPLICATION USAGE

33986 The *pthread_getcpuclockid()* function is part of the Thread CPU-Time Clocks option and need not
33987 be provided on all implementations.

33988 RATIONALE

33989 None.

33990 FUTURE DIRECTIONS

33991 None.

33992 SEE ALSO

33993 *clock_getcpuclockid()*, *clock_gettime()*, *timer_create()*, the Base Definitions volume of
33994 IEEE Std 1003.1-2001, <**pthread.h**>, <**time.h**>

33995 CHANGE HISTORY

33996 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

33997 In the SYNOPSIS, the inclusion of <**sys/types.h**> is no longer required.

33998 NAME

33999 pthread_getschedparam, pthread_setschedparam — dynamic thread scheduling parameters
 34000 access (**REALTIME THREADS**)

34001 SYNOPSIS

```
34002 THR TPS #include <pthread.h>
34003
34004     int pthread_getschedparam(pthread_t thread, int *restrict policy,
34005             struct sched_param *restrict param);
34006     int pthread_setschedparam(pthread_t thread, int policy,
34007             const struct sched_param *param);
```

34008 DESCRIPTION

34009 The *pthread_getschedparam()* and *pthread_setschedparam()* functions shall, respectively, get and set
 34010 the scheduling policy and parameters of individual threads within a multi-threaded process to
 34011 be retrieved and set. For SCHED_FIFO and SCHED_RR, the only required member of the
 34012 **sched_param** structure is the priority *sched_priority*. For SCHED_OTHER, the affected
 34013 scheduling parameters are implementation-defined.

34014 The *pthread_getschedparam()* function shall retrieve the scheduling policy and scheduling
 34015 parameters for the thread whose thread ID is given by *thread* and shall store those values in
 34016 *policy* and *param*, respectively. The priority value returned from *pthread_getschedparam()* shall be
 34017 the value specified by the most recent *pthread_setschedparam()*, *pthread_setschedprio()*, or
 34018 *pthread_create()* call affecting the target thread. It shall not reflect any temporary adjustments to
 34019 its priority as a result of any priority inheritance or ceiling functions. The *pthread_setschedparam()*
 34020 function shall set the scheduling policy and associated scheduling parameters for the thread
 34021 whose thread ID is given by *thread* to the *policy* and associated parameters provided in *policy*
 34022 and *param*, respectively.

34023 The *policy* parameter may have the value SCHED_OTHER, SCHED_FIFO, or SCHED_RR. The
 34024 scheduling parameters for the SCHED_OTHER policy are implementation-defined. The
 34025 SCHED_FIFO and SCHED_RR policies shall have a single scheduling parameter, *priority*.

34026 TSP If **_POSIX_THREAD_SPORADIC_SERVER** is defined, then the *policy* argument may have the
 34027 value SCHED_SPORADIC, with the exception for the *pthread_setschedparam()* function that if the
 34028 scheduling policy was not SCHED_SPORADIC at the time of the call, it is implementation-
 34029 defined whether the function is supported; in other words, the implementation need not allow
 34030 the application to dynamically change the scheduling policy to SCHED_SPORADIC. The
 34031 sporadic server scheduling policy has the associated parameters *sched_ss_low_priority*,
 34032 *sched_ss_repl_period*, *sched_ss_init_budget*, *sched_priority*, and *sched_ss_max_repl*. The specified
 34033 *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the
 34034 function to succeed; if it is not, then the function shall fail. The value of *sched_ss_max_repl* shall
 34035 be within the inclusive range [1,{SS_REPL_MAX}] for the function to succeed; if not, the function
 34036 shall fail.

34037 If the *pthread_setschedparam()* function fails, the scheduling parameters shall not be changed for
 34038 the target thread.

34039 RETURN VALUE

34040 If successful, the *pthread_getschedparam()* and *pthread_setschedparam()* functions shall return zero;
 34041 otherwise, an error number shall be returned to indicate the error.

34042 ERRORS

- 34043 The *pthread_getschedparam()* function may fail if:
- 34044 [ESRCH] The value specified by *thread* does not refer to an existing thread.
- 34045 The *pthread_setschedparam()* function may fail if:
- 34046 [EINVAL] The value specified by *policy* or one of the scheduling parameters associated with the scheduling policy *policy* is invalid.
- 34047
- 34048 [ENOTSUP] An attempt was made to set the policy or scheduling parameters to an unsupported value.
- 34049
- 34050 TSP [ENOTSUP] An attempt was made to dynamically change the scheduling policy to SCHED_SPORADIC, and the implementation does not support this change.
- 34051
- 34052 [EPERM] The caller does not have the appropriate permission to set either the scheduling parameters or the scheduling policy of the specified thread.
- 34053
- 34054 [EPERM] The implementation does not allow the application to modify one of the parameters to the value specified.
- 34055
- 34056 [ESRCH] The value specified by *thread* does not refer to a existing thread.
- 34057 These functions shall not return an error code of [EINTR].

34058 EXAMPLES

- 34059 None.

34060 APPLICATION USAGE

- 34061 None.

34062 RATIONALE

- 34063 None.

34064 FUTURE DIRECTIONS

- 34065 None.

34066 SEE ALSO

- 34067 *pthread_setschedprio()*, *sched_getparam()*, *sched_getscheduler()*, the Base Definitions volume of
34068 IEEE Std 1003.1-2001, <pthread.h>, <sched.h>

34069 CHANGE HISTORY

- 34070 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34071 Issue 6

- 34072 The *pthread_getschedparam()* and *pthread_setschedparam()* functions are marked as part of the
34073 Threads and Thread Execution Scheduling options.
- 34074 The [ENOSYS] error condition has been removed as stubs need not be provided if an
34075 implementation does not support the Thread Execution Scheduling option.
- 34076 The Open Group Corrigendum U026/2 is applied, correcting the prototype for the
34077 *pthread_setschedparam()* function so that its second argument is of type **int**.
- 34078 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.
- 34079 The **restrict** keyword is added to the *pthread_getschedparam()* prototype for alignment with the
34080 ISO/IEC 9899: 1999 standard.
- 34081 The Open Group Corrigendum U047/1 is applied.

34082
34083

IEEE PASC Interpretation 1003.1 #96 is applied, noting that priority values can also be set by a call to the *pthread_setschedprio()* function.

34084 NAME

34085 pthread_getspecific, pthread_setspecific — thread-specific data management

34086 SYNOPSIS

34087 THR #include <pthread.h>

```
34088        void *pthread_getspecific(pthread_key_t key);  
34089        int pthread_setspecific(pthread_key_t key, const void *value);  
34090
```

34091 DESCRIPTION

34092 The *pthread_getspecific()* function shall return the value currently bound to the specified *key* on behalf of the calling thread.

34094 The *pthread_setspecific()* function shall associate a thread-specific *value* with a *key* obtained via a previous call to *pthread_key_create()*. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

34098 The effect of calling *pthread_getspecific()* or *pthread_setspecific()* with a *key* value not obtained from *pthread_key_create()* or after *key* has been deleted with *pthread_key_delete()* is undefined.

34100 Both *pthread_getspecific()* and *pthread_setspecific()* may be called from a thread-specific data destructor function. A call to *pthread_getspecific()* for the thread-specific data key being destroyed shall return the value NULL, unless the value is changed (after the destructor starts) by a call to *pthread_setspecific()*. Calling *pthread_setspecific()* from a thread-specific data destructor routine may result either in lost storage (after at least PTHREAD_DESTRUCTOR_ITERATIONS attempts at destruction) or in an infinite loop.

34106 Both functions may be implemented as macros.

34107 RETURN VALUE

34108 The *pthread_getspecific()* function shall return the thread-specific data value associated with the given *key*. If no thread-specific data value is associated with *key*, then the value NULL shall be returned.

34111 If successful, the *pthread_setspecific()* function shall return zero; otherwise, an error number shall be returned to indicate the error.

34113 ERRORS

34114 No errors are returned from *pthread_getspecific()*.

34115 The *pthread_setspecific()* function shall fail if:

34116 [ENOMEM] Insufficient memory exists to associate the non-NUL value with the key. 2

34117 The *pthread_setspecific()* function may fail if:

34118 [EINVAL] The *key* value is invalid.

34119 These functions shall not return an error code of [EINTR].

34120 EXAMPLES

34121 None.

34122 APPLICATION USAGE

34123 None.

34124 RATIONALE

34125 Performance and ease-of-use of *pthread_getspecific()* are critical for functions that rely on
34126 maintaining state in thread-specific data. Since no errors are required to be detected by it, and
34127 since the only error that could be detected is the use of an invalid key, the function to
34128 *pthread_getspecific()* has been designed to favor speed and simplicity over error reporting.

34129 FUTURE DIRECTIONS

34130 None.

34131 SEE ALSO

34132 *pthread_key_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

34133 CHANGE HISTORY

34134 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34135 Issue 6

34136 The *pthread_getspecific()* and *pthread_setspecific()* functions are marked as part of the Threads
34137 option.

34138 IEEE PASC Interpretation 1003.1c #3 (Part 6) is applied, updating the DESCRIPTION.

34139 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/96 is applied, updating the ERRORS 2
34140 section so that the [ENOMEM] error case is changed from “to associate the value with the key” 2
34141 to “to associate the non-NUL value with the key”. 2

34142 NAME

34143 pthread_join — wait for thread termination

34144 SYNOPSIS

34145 THR #include <pthread.h>

34146 int pthread_join(pthread_t *thread*, void ***value_ptr*);

34147

34148 DESCRIPTION

34149 The *pthread_join()* function shall suspend execution of the calling thread until the target *thread*
 34150 terminates, unless the target *thread* has already terminated. On return from a successful
 34151 *pthread_join()* call with a non-NULL *value_ptr* argument, the value passed to *pthread_exit()* by
 34152 the terminating thread shall be made available in the location referenced by *value_ptr*. When a
 34153 *pthread_join()* returns successfully, the target thread has been terminated. The results of multiple
 34154 simultaneous calls to *pthread_join()* specifying the same target thread are undefined. If the
 34155 thread calling *pthread_join()* is canceled, then the target thread shall not be detached.

34156 It is unspecified whether a thread that has exited but remains unjoined counts against
 34157 {PTHREAD_THREADS_MAX}.

34158 RETURN VALUE

34159 If successful, the *pthread_join()* function shall return zero; otherwise, an error number shall be
 34160 returned to indicate the error.

34161 ERRORS

34162 The *pthread_join()* function shall fail if:

34163 [ESRCH] No thread could be found corresponding to that specified by the given thread ID. 2

34165 The *pthread_join()* function may fail if:

34166 [EDEADLK] A deadlock was detected or the value of *thread* specifies the calling thread. 2

34167 [EINVAL] The value specified by *thread* does not refer to a joinable thread. 2

34168 The *pthread_join()* function shall not return an error code of [EINTR].

34169 EXAMPLES

34170 An example of thread creation and deletion follows:

```
34171     typedef struct {
34172         int *ar;
34173         long n;
34174     } subarray;
34175
34176     void *
34177     incer(void *arg)
34178     {
34179         long i;
34180
34181         for (i = 0; i < ((subarray *)arg)->n; i++)
34182             ((subarray *)arg)->ar[i]++;
34183     }
34184
34185     int main(void)
34186     {
34187         int           ar[1000000];
34188         pthread_t    th1, th2;
```

```
34186     subarray    sb1, sb2;
34187     sb1.ar = &ar[0];
34188     sb1.n  = 500000;
34189     (void) pthread_create(&th1, NULL, incer, &sb1);
34190     sb2.ar = &ar[500000];
34191     sb2.n  = 500000;
34192     (void) pthread_create(&th2, NULL, incer, &sb2);
34193     (void) pthread_join(th1, NULL);
34194     (void) pthread_join(th2, NULL);
34195     return 0;
34196 }
```

34197 APPLICATION USAGE

34198 None.

34199 RATIONALE

34200 The *pthread_join()* function is a convenience that has proven useful in multi-threaded
34201 applications. It is true that a programmer could simulate this function if it were not provided by
34202 passing extra state as part of the argument to the *start_routine()*. The terminating thread would
34203 set a flag to indicate termination and broadcast a condition that is part of that state; a joining
34204 thread would wait on that condition variable. While such a technique would allow a thread to
34205 wait on more complex conditions (for example, waiting for multiple threads to terminate),
34206 waiting on individual thread termination is considered widely useful. Also, including the
34207 *pthread_join()* function in no way precludes a programmer from coding such complex waits.
34208 Thus, while not a primitive, including *pthread_join()* in this volume of IEEE Std 1003.1-2001 was
34209 considered valuable.

34210 The *pthread_join()* function provides a simple mechanism allowing an application to wait for a
34211 thread to terminate. After the thread terminates, the application may then choose to clean up
34212 resources that were used by the thread. For instance, after *pthread_join()* returns, any
34213 application-provided stack storage could be reclaimed.

34214 The *pthread_join()* or *pthread_detach()* function should eventually be called for every thread that
34215 is created with the *detachstate* attribute set to PTHREAD_CREATE_JOINABLE so that storage
34216 associated with the thread may be reclaimed.

34217 The interaction between *pthread_join()* and cancellation is well-defined for the following
34218 reasons:

- 34219 • The *pthread_join()* function, like all other non-async-cancel-safe functions, can only be called
34220 with deferred cancelability type.
- 34221 • Cancellation cannot occur in the disabled cancelability state.

34222 Thus, only the default cancelability state need be considered. As specified, either the
34223 *pthread_join()* call is canceled, or it succeeds, but not both. The difference is obvious to the
34224 application, since either a cancellation handler is run or *pthread_join()* returns. There are no race
34225 conditions since *pthread_join()* was called in the deferred cancelability state.

34226 FUTURE DIRECTIONS

34227 None.

34228 SEE ALSO

34229 *pthread_create(), wait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

34230 CHANGE HISTORY

34231 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34232 Issue 6

34233 The `pthread_join()` function is marked as part of the Threads option.

34234 IEEE Std 1003.1-2001/Cor 2-2004, XSH/TC2/D6/97 is applied, updating the ERRORS section so 2
34235 that the [EINVAL] error is made optional and the words “the implementation has detected” are 2
34236 removed from it. 2

34237 NAME

34238 pthread_key_create — thread-specific data key creation

34239 SYNOPSIS

34240 THR #include <pthread.h>

```
34241        int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```

34242

34243 DESCRIPTION

34244 The *pthread_key_create()* function shall create a thread-specific data key visible to all threads in
34245 the process. Key values provided by *pthread_key_create()* are opaque objects used to locate
34246 thread-specific data. Although the same key value may be used by different threads, the values
34247 bound to the key by *pthread_setspecific()* are maintained on a per-thread basis and persist for the
34248 life of the calling thread.

34249 Upon key creation, the value NULL shall be associated with the new key in all active threads.
34250 Upon thread creation, the value NULL shall be associated with all defined keys in the new
34251 thread.

34252 An optional destructor function may be associated with each key value. At thread exit, if a key
34253 value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with
34254 that key, the value of the key is set to NULL, and then the function pointed to is called with the
34255 previously associated value as its sole argument. The order of destructor calls is unspecified if
34256 more than one destructor exists for a thread when it exits.

34257 If, after all the destructors have been called for all non-NULL values with associated destructors,
34258 there are still some non-NULL values with associated destructors, then the process is repeated.
34259 If, after at least {PTHREAD_DESTRUCTOR_ITERATIONS} iterations of destructor calls for
34260 outstanding non-NULL values, there are still some non-NULL values with associated
34261 destructors, implementations may stop calling destructors, or they may continue calling
34262 destructors until no non-NULL values with associated destructors exist, even though this might
34263 result in an infinite loop.

34264 RETURN VALUE

34265 If successful, the *pthread_key_create()* function shall store the newly created key value at *key and
34266 shall return zero. Otherwise, an error number shall be returned to indicate the error.

34267 ERRORS

34268 The *pthread_key_create()* function shall fail if:

34269 [EAGAIN] The system lacked the necessary resources to create another thread-specific
34270 data key, or the system-imposed limit on the total number of keys per process
34271 {PTHREAD_KEYS_MAX} has been exceeded.

34272 [ENOMEM] Insufficient memory exists to create the key.

34273 The *pthread_key_create()* function shall not return an error code of [EINTR].

34274 EXAMPLES

34275 The following example demonstrates a function that initializes a thread-specific data key when
34276 it is first called, and associates a thread-specific object with each calling thread, initializing this
34277 object when necessary.

```
34278 static pthread_key_t key;
34279 static pthread_once_t key_once = PTHREAD_ONCE_INIT;
34280
34281 static void
34282 make_key()
34283 {
34284     (void) pthread_key_create(&key, NULL);
34285 }
34286
34287 func()
34288 {
34289     void *ptr;
34290
34291     (void) pthread_once(&key_once, make_key);
34292     if ((ptr = pthread_getspecific(key)) == NULL) {
34293         ptr = malloc(OBJECT_SIZE);
34294         ...
34295         (void) pthread_setspecific(key, ptr);
34296     }
34297     ...
34298 }
```

34299 Note that the key has to be initialized before *pthread_getspecific()* or *pthread_setspecific()* can be
34300 used. The *pthread_key_create()* call could either be explicitly made in a module initialization
34301 routine, or it can be done implicitly by the first call to a module as in this example. Any attempt
34302 to use the key before it is initialized is a programming error, making the code below incorrect.

```
34300 static pthread_key_t key;
34301 func()
34302 {
34303     void *ptr;
34304
34305     /* KEY NOT INITIALIZED!!! THIS WON'T WORK!!! */
34306     if ((ptr = pthread_getspecific(key)) == NULL &&
34307         pthread_setspecific(key, NULL) != 0) {
34308         pthread_key_create(&key, NULL);
34309         ...
34310     }
34311 }
```

34311 APPLICATION USAGE

34312 None.

34313 **RATIONALE**34314 **Destructor Functions**

34315 Normally, the value bound to a key on behalf of a particular thread is a pointer to storage
34316 allocated dynamically on behalf of the calling thread. The destructor functions specified with
34317 *pthread_key_create()* are intended to be used to free this storage when the thread exits. Thread
34318 cancellation cleanup handlers cannot be used for this purpose because thread-specific data may
34319 persist outside the lexical scope in which the cancellation cleanup handlers operate.

34320 If the value associated with a key needs to be updated during the lifetime of the thread, it may
34321 be necessary to release the storage associated with the old value before the new value is bound.
34322 Although the *pthread_setspecific()* function could do this automatically, this feature is not needed
34323 often enough to justify the added complexity. Instead, the programmer is responsible for freeing
34324 the stale storage:

```
34325 pthread_getspecific(key, &old);  
34326 new = allocate();  
34327 destructor(old);  
34328 pthread_setspecific(key, new);
```

34329 **Note:** The above example could leak storage if run with asynchronous cancellation enabled. No such
34330 problems occur in the default cancellation state if no cancellation points occur between the get
34331 and set.

34332 There is no notion of a destructor-safe function. If an application does not call *pthread_exit()*
34333 from a signal handler, or if it blocks any signal whose handler may call *pthread_exit()* while
34334 calling async-unsafe functions, all functions may be safely called from destructors.

34335 **Non-Idempotent Data Key Creation**

34336 There were requests to make *pthread_key_create()* idempotent with respect to a given *key* address
34337 parameter. This would allow applications to call *pthread_key_create()* multiple times for a given
34338 *key* address and be guaranteed that only one key would be created. Doing so would require the
34339 key value to be previously initialized (possibly at compile time) to a known null value and
34340 would require that implicit mutual-exclusion be performed based on the address and contents of
34341 the *key* parameter in order to guarantee that exactly one key would be created.

34342 Unfortunately, the implicit mutual-exclusion would not be limited to only *pthread_key_create()*.
34343 On many implementations, implicit mutual-exclusion would also have to be performed by
34344 *pthread_getspecific()* and *pthread_setspecific()* in order to guard against using incompletely stored
34345 or not-yet-visible key values. This could significantly increase the cost of important operations,
34346 particularly *pthread_getspecific()*.

34347 Thus, this proposal was rejected. The *pthread_key_create()* function performs no implicit
34348 synchronization. It is the responsibility of the programmer to ensure that it is called exactly once
34349 per key before use of the key. Several straightforward mechanisms can already be used to
34350 accomplish this, including calling explicit module initialization functions, using mutexes, and
34351 using *pthread_once()*. This places no significant burden on the programmer, introduces no
34352 possibly confusing *ad hoc* implicit synchronization mechanism, and potentially allows
34353 commonly used thread-specific data operations to be more efficient.

34354 **FUTURE DIRECTIONS**

34355 None.

34356 SEE ALSO

34357 *pthread_getspecific()*, *pthread_key_delete()*, the Base Definitions volume of IEEE Std 1003.1-2001,
34358 <*pthread.h*>

34359 CHANGE HISTORY

34360 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34361 Issue 6

34362 The *pthread_key_create()* function is marked as part of the Threads option.
34363 IEEE PASC Interpretation 1003.1c #8 is applied, updating the DESCRIPTION.

34364 NAME

34365 *pthread_key_delete* — thread-specific data key deletion

34366 SYNOPSIS

34367 *THR* `#include <pthread.h>`

34368 `int pthread_key_delete(pthread_key_t key);`

34369

34370 DESCRIPTION

34371 The *pthread_key_delete()* function shall delete a thread-specific data key previously returned by
34372 *pthread_key_create()*. The thread-specific data values associated with *key* need not be NULL at
34373 the time *pthread_key_delete()* is called. It is the responsibility of the application to free any
34374 application storage or perform any cleanup actions for data structures related to the deleted key
34375 or associated thread-specific data in any threads; this cleanup can be done either before or after
34376 *pthread_key_delete()* is called. Any attempt to use *key* following the call to *pthread_key_delete()*
34377 results in undefined behavior.

34378 The *pthread_key_delete()* function shall be callable from within destructor functions. No
34379 destructor functions shall be invoked by *pthread_key_delete()*. Any destructor function that may
34380 have been associated with *key* shall no longer be called upon thread exit.

34381 RETURN VALUE

34382 If successful, the *pthread_key_delete()* function shall return zero; otherwise, an error number shall
34383 be returned to indicate the error.

34384 ERRORS

34385 The *pthread_key_delete()* function may fail if:

34386 [EINVAL] The *key* value is invalid.

34387 The *pthread_key_delete()* function shall not return an error code of [EINTR].

34388 EXAMPLES

34389 None.

34390 APPLICATION USAGE

34391 None.

34392 RATIONALE

34393 A thread-specific data key deletion function has been included in order to allow the resources
34394 associated with an unused thread-specific data key to be freed. Unused thread-specific data keys
34395 can arise, among other scenarios, when a dynamically loaded module that allocated a key is
34396 unloaded.

34397 Conforming applications are responsible for performing any cleanup actions needed for data
34398 structures associated with the key to be deleted, including data referenced by thread-specific
34399 data values. No such cleanup is done by *pthread_key_delete()*. In particular, destructor functions
34400 are not called. There are several reasons for this division of responsibility:

- 34401 1. The associated destructor functions used to free thread-specific data at thread exit time are
34402 only guaranteed to work correctly when called in the thread that allocated the thread-
34403 specific data. (Destructors themselves may utilize thread-specific data.) Thus, they cannot
34404 be used to free thread-specific data in other threads at key deletion time. Attempting to
34405 have them called by other threads at key deletion time would require other threads to be
34406 asynchronously interrupted. But since interrupted threads could be in an arbitrary state,
34407 including holding locks necessary for the destructor to run, this approach would fail. In
34408 general, there is no safe mechanism whereby an implementation could free thread-specific
34409 data at key deletion time.

34410 2. Even if there were a means of safely freeing thread-specific data associated with keys to be
34411 deleted, doing so would require that implementations be able to enumerate the threads
34412 with non-NUL data and potentially keep them from creating more thread-specific data
34413 while the key deletion is occurring. This special case could cause extra synchronization in
34414 the normal case, which would otherwise be unnecessary.

34415 For an application to know that it is safe to delete a key, it has to know that all the threads that
34416 might potentially ever use the key do not attempt to use it again. For example, it could know this
34417 if all the client threads have called a cleanup procedure declaring that they are through with the
34418 module that is being shut down, perhaps by setting a reference count to zero.

34419 **FUTURE DIRECTIONS**

34420 None.

34421 **SEE ALSO**

34422 *pthread_key_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

34423 **CHANGE HISTORY**

34424 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34425 **Issue 6**

34426 The *pthread_key_delete()* function is marked as part of the Threads option.

34427 NAME

34428 *pthread_kill* — send a signal to a thread

34429 SYNOPSIS

34430 **THR** `#include <signal.h>`

34431 `int pthread_kill(pthread_t thread, int sig);`

34432

34433 DESCRIPTION

34434 The *pthread_kill()* function shall request that a signal be delivered to the specified thread.

34435 As in *kill()*, if *sig* is zero, error checking shall be performed but no signal shall actually be sent.

34436 RETURN VALUE

34437 Upon successful completion, the function shall return a value of zero. Otherwise, the function shall return an error number. If the *pthread_kill()* function fails, no signal shall be sent.

34439 ERRORS

34440 The *pthread_kill()* function shall fail if:

34441 [ESRCH] No thread could be found corresponding to that specified by the given thread ID.

34443 [EINVAL] The value of the *sig* argument is an invalid or unsupported signal number.

34444 The *pthread_kill()* function shall not return an error code of [EINTR].

34445 EXAMPLES

34446 None.

34447 APPLICATION USAGE

34448 The *pthread_kill()* function provides a mechanism for asynchronously directing a signal at a thread in the calling process. This could be used, for example, by one thread to affect broadcast delivery of a signal to a set of threads.

34451 Note that *pthread_kill()* only causes the signal to be handled in the context of the given thread; the signal action (termination or stopping) affects the process as a whole.

34453 RATIONALE

34454 None.

34455 FUTURE DIRECTIONS

34456 None.

34457 SEE ALSO

34458 *kill()*, *pthread_self()*, *raise()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>

34459 CHANGE HISTORY

34460 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34461 Issue 6

34462 The *pthread_kill()* function is marked as part of the Threads option.

34463 The APPLICATION USAGE section is added.

34464 NAME

34465 pthread_mutex_destroy, pthread_mutex_init — destroy and initialize a mutex

34466 SYNOPSIS

34467 THR #include <pthread.h>

```
34468     int pthread_mutex_destroy(pthread_mutex_t *mutex);
34469     int pthread_mutex_init(pthread_mutex_t *restrict mutex,
34470                        const pthread_mutexattr_t *restrict attr);
34471     pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
34472
```

34473 DESCRIPTION

34474 The *pthread_mutex_destroy()* function shall destroy the mutex object referenced by *mutex*; the
 34475 mutex object becomes, in effect, uninitialized. An implementation may cause
 34476 *pthread_mutex_destroy()* to set the object referenced by *mutex* to an invalid value. A destroyed
 34477 mutex object can be reinitialized using *pthread_mutex_init()*; the results of otherwise referencing
 34478 the object after it has been destroyed are undefined.

34479 It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked
 34480 mutex results in undefined behavior.

34481 The *pthread_mutex_init()* function shall initialize the mutex referenced by *mutex* with attributes
 34482 specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the
 34483 same as passing the address of a default mutex attributes object. Upon successful initialization,
 34484 the state of the mutex becomes initialized and unlocked.

34485 Only *mutex* itself may be used for performing synchronization. The result of referring to copies
 34486 of *mutex* in calls to *pthread_mutex_lock()*, *pthread_mutex_trylock()*, *pthread_mutex_unlock()*, and
 34487 *pthread_mutex_destroy()* is undefined.

34488 Attempting to initialize an already initialized mutex results in undefined behavior.

34489 In cases where default mutex attributes are appropriate, the macro
 34490 PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated.
 34491 The effect shall be equivalent to dynamic initialization by a call to *pthread_mutex_init()* with
 34492 parameter *attr* specified as NULL, except that no error checks are performed.

34493 RETURN VALUE

34494 If successful, the *pthread_mutex_destroy()* and *pthread_mutex_init()* functions shall return zero;
 34495 otherwise, an error number shall be returned to indicate the error.

34496 The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed
 34497 immediately at the beginning of processing for the function and shall cause an error return prior
 34498 to modifying the state of the mutex specified by *mutex*.

34499 ERRORS

34500 The *pthread_mutex_destroy()* function may fail if:

34501 [EBUSY] The implementation has detected an attempt to destroy the object referenced
 34502 by *mutex* while it is locked or referenced (for example, while being used in a
 34503 *pthread_cond_timedwait()* or *pthread_cond_wait()*) by another thread.

34504 [EINVAL] The value specified by *mutex* is invalid.

34505 The *pthread_mutex_init()* function shall fail if:

34506 [EAGAIN] The system lacked the necessary resources (other than memory) to initialize
 34507 another mutex.

34508 [ENOMEM] Insufficient memory exists to initialize the mutex.
34509 [EPERM] The caller does not have the privilege to perform the operation.
34510 The *pthread_mutex_init()* function may fail if:
34511 [EBUSY] The implementation has detected an attempt to reinitialize the object
34512 referenced by *mutex*, a previously initialized, but not yet destroyed, mutex.
34513 [EINVAL] The value specified by *attr* is invalid.
34514 These functions shall not return an error code of [EINTR].

34515 EXAMPLES

34516 None.

34517 APPLICATION USAGE

34518 None.

34519 RATIONALE

34520 Alternate Implementations Possible

34521 This volume of IEEE Std 1003.1-2001 supports several alternative implementations of mutexes.
34522 An implementation may store the lock directly in the object of type *pthread_mutex_t*.
34523 Alternatively, an implementation may store the lock in the heap and merely store a pointer,
34524 handle, or unique ID in the mutex object. Either implementation has advantages or may be
34525 required on certain hardware configurations. So that portable code can be written that is
34526 invariant to this choice, this volume of IEEE Std 1003.1-2001 does not define assignment or
34527 equality for this type, and it uses the term “initialize” to reinforce the (more restrictive) notion
34528 that the lock may actually reside in the mutex object itself.

34529 Note that this precludes an over-specification of the type of the mutex or condition variable and
34530 motivates the opaqueness of the type.

34531 An implementation is permitted, but not required, to have *pthread_mutex_destroy()* store an
34532 illegal value into the mutex. This may help detect erroneous programs that try to lock (or
34533 otherwise reference) a mutex that has already been destroyed.

34534 Tradeoff Between Error Checks and Performance Supported

34535 Many of the error checks were made optional in order to let implementations trade off
34536 performance versus degree of error checking according to the needs of their specific applications
34537 and execution environment. As a general rule, errors or conditions caused by the system (such as
34538 insufficient memory) always need to be reported, but errors due to an erroneously coded
34539 application (such as failing to provide adequate synchronization to prevent a mutex from being
34540 deleted while in use) are made optional.

34541 A wide range of implementations is thus made possible. For example, an implementation
34542 intended for application debugging may implement all of the error checks, but an
34543 implementation running a single, provably correct application under very tight performance
34544 constraints in an embedded computer might implement minimal checks. An implementation
34545 might even be provided in two versions, similar to the options that compilers provide: a full-
34546 checking, but slower version; and a limited-checking, but faster version. To forbid this
34547 optionality would be a disservice to users.

34548 By carefully limiting the use of “undefined behavior” only to things that an erroneous (badly
34549 coded) application might do, and by defining that resource-not-available errors are mandatory,
34550 this volume of IEEE Std 1003.1-2001 ensures that a fully-conforming application is portable

34551 across the full range of implementations, while not forcing all implementations to add overhead
34552 to check for numerous things that a correct program never does.

34553 **Why No Limits are Defined**

34554 Defining symbols for the maximum number of mutexes and condition variables was considered
34555 but rejected because the number of these objects may change dynamically. Furthermore, many
34556 implementations place these objects into application memory; thus, there is no explicit
34557 maximum.

34558 **Static Initializers for Mutexes and Condition Variables**

34559 Providing for static initialization of statically allocated synchronization objects allows modules
34560 with private static synchronization variables to avoid runtime initialization tests and overhead.
34561 Furthermore, it simplifies the coding of self-initializing modules. Such modules are common in
34562 C libraries, where for various reasons the design calls for self-initialization instead of requiring
34563 an explicit module initialization function to be called. An example use of static initialization
34564 follows.

34565 Without static initialization, a self-initializing routine *foo()* might look as follows:

```
34566 static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
34567 static pthread_mutex_t foo_mutex;
34568
34569 void foo_init()
34570 {
34571     pthread_mutex_init(&foo_mutex, NULL);
34572 }
34573
34574 void foo()
34575 {
34576     pthread_once(&foo_once, foo_init);
34577     pthread_mutex_lock(&foo_mutex);
34578     /* Do work. */
34579     pthread_mutex_unlock(&foo_mutex);
34580 }
```

34581 With static initialization, the same routine could be coded as follows:

```
34582 static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;
34583
34584 void foo()
34585 {
34586     pthread_mutex_lock(&foo_mutex);
34587     /* Do work. */
34588     pthread_mutex_unlock(&foo_mutex);
34589 }
```

34590 Note that the static initialization both eliminates the need for the initialization test inside
34591 *pthread_once()* and the fetch of *&foo_mutex* to learn the address to be passed to
34592 *pthread_mutex_lock()* or *pthread_mutex_unlock()*.

34593 Thus, the C code written to initialize static objects is simpler on all systems and is also faster on a
34594 large class of systems; those where the (entire) synchronization object can be stored in
34595 application memory.

34596 Yet the locking performance question is likely to be raised for machines that require mutexes to
34597 be allocated out of special memory. Such machines actually have to have mutexes and possibly

34595 condition variables contain pointers to the actual hardware locks. For static initialization to work
34596 on such machines, *pthread_mutex_lock()* also has to test whether or not the pointer to the actual
34597 lock has been allocated. If it has not, *pthread_mutex_lock()* has to initialize it before use. The
34598 reservation of such resources can be made when the program is loaded, and hence return codes
34599 have not been added to mutex locking and condition variable waiting to indicate failure to
34600 complete initialization.

34601 This runtime test in *pthread_mutex_lock()* would at first seem to be extra work; an extra test is
34602 required to see whether the pointer has been initialized. On most machines this would actually
34603 be implemented as a fetch of the pointer, testing the pointer against zero, and then using the
34604 pointer if it has already been initialized. While the test might seem to add extra work, the extra
34605 effort of testing a register is usually negligible since no extra memory references are actually
34606 done. As more and more machines provide caches, the real expenses are memory references, not
34607 instructions executed.

34608 Alternatively, depending on the machine architecture, there are often ways to eliminate *all*
34609 overhead in the most important case: on the lock operations that occur *after* the lock has been
34610 initialized. This can be done by shifting more overhead to the less frequent operation:
34611 initialization. Since out-of-line mutex allocation also means that an address has to be
34612 dereferenced to find the actual lock, one technique that is widely applicable is to have static
34613 initialization store a bogus value for that address; in particular, an address that causes a machine
34614 fault to occur. When such a fault occurs upon the first attempt to lock such a mutex, validity
34615 checks can be done, and then the correct address for the actual lock can be filled in. Subsequent
34616 lock operations incur no extra overhead since they do not “fault”. This is merely one technique
34617 that can be used to support static initialization, while not adversely affecting the performance of
34618 lock acquisition. No doubt there are other techniques that are highly machine-dependent.

34619 The locking overhead for machines doing out-of-line mutex allocation is thus similar for
34620 modules being implicitly initialized, where it is improved for those doing mutex allocation
34621 entirely inline. The inline case is thus made much faster, and the out-of-line case is not
34622 significantly worse.

34623 Besides the issue of locking performance for such machines, a concern is raised that it is possible
34624 that threads would serialize contending for initialization locks when attempting to finish
34625 initializing statically allocated mutexes. (Such finishing would typically involve taking an
34626 internal lock, allocating a structure, storing a pointer to the structure in the mutex, and releasing
34627 the internal lock.) First, many implementations would reduce such serialization by hashing on
34628 the mutex address. Second, such serialization can only occur a bounded number of times. In
34629 particular, it can happen at most as many times as there are statically allocated synchronization
34630 objects. Dynamically allocated objects would still be initialized via *pthread_mutex_init()* or
34631 *pthread_cond_init()*.

34632 Finally, if none of the above optimization techniques for out-of-line allocation yields sufficient
34633 performance for an application on some implementation, the application can avoid static
34634 initialization altogether by explicitly initializing all synchronization objects with the
34635 corresponding *pthread_*_init()* functions, which are supported by all implementations. An
34636 implementation can also document the tradeoffs and advise which initialization technique is
34637 more efficient for that particular implementation.

34638 **Destroying Mutexes**

34639 A mutex can be destroyed immediately after it is unlocked. For example, consider the following
34640 code:

```
34641       struct obj {
34642        pthread_mutex_t om;
34643        int refcnt;
34644        ...
34645       } ;

34646       obj_done(struct obj *op)
34647       {
34648        pthread_mutex_lock(&op->om);
34649        if (--op->refcnt == 0) {
34650           pthread_mutex_unlock(&op->om);
34651           (A)    pthread_mutex_destroy(&op->om);
34652           (B)    free(op);
34653           } else
34654           (C)    pthread_mutex_unlock(&op->om);
34655       }
```

34656 In this case *obj* is reference counted and *obj_done()* is called whenever a reference to the object is
34657 dropped. Implementations are required to allow an object to be destroyed and freed and
34658 potentially unmapped (for example, lines A and B) immediately after the object is unlocked (line
34659 C).

34660 **FUTURE DIRECTIONS**

34661 None.

34662 **SEE ALSO**

34663 *pthread_mutex_getprioceiling()*, *pthread_mutex_lock()*, *pthread_mutex_timedlock()*,
34664 *pthread_mutexattr_getpshared()*, the Base Definitions volume of IEEE Std 1003.1-2001,
34665 <*pthread.h*>

34666 **CHANGE HISTORY**

34667 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34668 **Issue 6**

34669 The *pthread_mutex_destroy()* and *pthread_mutex_init()* functions are marked as part of the
34670 Threads option.

34671 The *pthread_mutex_timedlock()* function is added to the SEE ALSO section for alignment with
34672 IEEE Std 1003.1d-1999.

34673 IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

34674 The **restrict** keyword is added to the *pthread_mutex_init()* prototype for alignment with the
34675 ISO/IEC 9899:1999 standard.

34676 NAME

34677 *pthread_mutex_getprioceiling*, *pthread_mutex_setprioceiling* — get and set the priority ceiling
 34678 of a mutex (**REALTIME THREADS**)

34679 SYNOPSIS

```
34680 THR TPP #include <pthread.h>
34681     int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
34682             int *restrict prioceiling);
34683     int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
34684             int prioceiling, int *restrict old_ceiling);
34685
```

34686 DESCRIPTION

34687 The *pthread_mutex_getprioceiling*() function shall return the current priority ceiling of the mutex.

34688 The *pthread_mutex_setprioceiling*() function shall either lock the mutex if it is unlocked, or block
 34689 until it can successfully lock the mutex, then it shall change the mutex's priority ceiling and
 34690 release the mutex. When the change is successful, the previous value of the priority ceiling shall
 34691 be returned in *old_ceiling*. The process of locking the mutex need not adhere to the priority
 34692 protect protocol.

34693 If the *pthread_mutex_setprioceiling*() function fails, the mutex priority ceiling shall not be
 34694 changed.

34695 RETURN VALUE

34696 If successful, the *pthread_mutex_getprioceiling*() and *pthread_mutex_setprioceiling*() functions shall
 34697 return zero; otherwise, an error number shall be returned to indicate the error.

34698 ERRORS

34699 The *pthread_mutex_getprioceiling*() and *pthread_mutex_setprioceiling*() functions may fail if:

34700 [EINVAL]	The priority requested by <i>prioceiling</i> is out of range.
34701 [EINVAL]	The value specified by <i>mutex</i> does not refer to a currently existing mutex.
34702 [EPERM]	The caller does not have the privilege to perform the operation.

34703 These functions shall not return an error code of [EINTR].

34704 EXAMPLES

34705 None.

34706 APPLICATION USAGE

34707 None.

34708 RATIONALE

34709 None.

34710 FUTURE DIRECTIONS

34711 None.

34712 SEE ALSO

34713 *pthread_mutex_destroy*(), *pthread_mutex_lock*(), *pthread_mutex_timedlock*(), the Base Definitions
 34714 volume of IEEE Std 1003.1-2001, <pthread.h>

34715 CHANGE HISTORY

34716 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

34717 Marked as part of the Realtime Threads Feature Group.

34718 Issue 6

- 34719 The *pthread_mutex_getprioceiling()* and *pthread_mutex_setprioceiling()* functions are marked as part of the Threads and Thread Priority Protection options.
- 34720
- 34721 The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Priority Protection option.
- 34722
- 34723 The [ENOSYS] error denoting non-support of the priority ceiling protocol for mutexes has been removed. This is because if the implementation provides the functions (regardless of whether *_POSIX_PTHREAD_PRIO_PROTECT* is defined), they must function as in the DESCRIPTION and therefore the priority ceiling protocol for mutexes is supported.
- 34724
- 34725
- 34726
- 34727 The *pthread_mutex_timedlock()* function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
- 34728
- 34729 The **restrict** keyword is added to the *pthread_mutex_getprioceiling()* and *pthread_mutex_setprioceiling()* prototypes for alignment with the ISO/IEC 9899: 1999 standard.
- 34730

34731 **NAME**34732 **pthread_mutex_init** — initialize a mutex34733 **SYNOPSIS**34734 **THR** `#include <pthread.h>`

```
34735       int pthread_mutex_init(pthread_mutex_t *restrict mutex,
34736                 const pthread_mutexattr_t *restrict attr);
34737       pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
34738
```

34739 **DESCRIPTION**34740 Refer to *pthread_mutex_destroy()*.

34741 NAME

34742 pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a
 34743 mutex

34744 SYNOPSIS

34745 THR #include <pthread.h>
 34746 int pthread_mutex_lock(pthread_mutex_t *mutex);
 34747 int pthread_mutex_trylock(pthread_mutex_t *mutex);
 34748 int pthread_mutex_unlock(pthread_mutex_t *mutex);
 34749

34750 DESCRIPTION

34751 The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock()*. If the
 34752 mutex is already locked, the calling thread shall block until the mutex becomes available. This
 34753 operation shall return with the mutex object referenced by *mutex* in the locked state with the
 34754 calling thread as its owner.

34755 XSI If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection shall not be provided.
 34756 Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it
 34757 has not locked or a mutex which is unlocked, undefined behavior results.

34758 If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking shall be provided.
 34759 If a thread attempts to relock a mutex that it has already locked, an error shall be returned. If a
 34760 thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error
 34761 shall be returned.

34762 If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex shall maintain the
 34763 concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock
 34764 count shall be set to one. Every time a thread relocks this mutex, the lock count shall be
 34765 incremented by one. Each time the thread unlocks the mutex, the lock count shall be
 34766 decremented by one. When the lock count reaches zero, the mutex shall become available for
 34767 other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex
 34768 which is unlocked, an error shall be returned.

34769 If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex
 34770 results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling
 34771 thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in
 34772 undefined behavior.

34773 The *pthread_mutex_trylock()* function shall be equivalent to *pthread_mutex_lock()*, except that if
 34774 the mutex object referenced by *mutex* is currently locked (by any thread, including the current
 34775 thread), the call shall return immediately. If the mutex type is PTHREAD_MUTEX_RECURSIVE
 34776 and the mutex is currently owned by the calling thread, the mutex lock count shall be
 34777 incremented by one and the *pthread_mutex_trylock()* function shall immediately return success.

34778 XSI The *pthread_mutex_unlock()* function shall release the mutex object referenced by *mutex*. The
 34779 manner in which a mutex is released is dependent upon the mutex's type attribute. If there are
 34780 threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock()* is called,
 34781 resulting in the mutex becoming available, the scheduling policy shall determine which thread
 34782 shall acquire the mutex.

34783 XSI (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex shall become available
 34784 when the count reaches zero and the calling thread no longer has any locks on this mutex.)

34785 If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the
 34786 thread shall resume waiting for the mutex as if it was not interrupted.

34787 RETURN VALUE

34788 If successful, the *pthread_mutex_lock()* and *pthread_mutex_unlock()* functions shall return zero;
34789 otherwise, an error number shall be returned to indicate the error.

34790 The *pthread_mutex_trylock()* function shall return zero if a lock on the mutex object referenced by
34791 *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

34792 ERRORS

34793 The *pthread_mutex_lock()* and *pthread_mutex_trylock()* functions shall fail if:

34794 [EINVAL] The *mutex* was created with the protocol attribute having the value
34795 PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than
34796 the mutex's current priority ceiling.

34797 The *pthread_mutex_trylock()* function shall fail if:

34798 [EBUSY] The *mutex* could not be acquired because it was already locked.

34799 The *pthread_mutex_lock()*, *pthread_mutex_trylock()*, and *pthread_mutex_unlock()* functions may
34800 fail if:

34801 [EINVAL] The value specified by *mutex* does not refer to an initialized mutex object.

34802 XSI [EAGAIN] The mutex could not be acquired because the maximum number of recursive
34803 locks for *mutex* has been exceeded.

34804 The *pthread_mutex_lock()* function may fail if:

34805 [EDEADLK] A deadlock condition was detected or the current thread already owns the 2
34806 mutex. 2

34807 The *pthread_mutex_unlock()* function may fail if:

34808 [EPERM] The current thread does not own the mutex.

34809 These functions shall not return an error code of [EINTR].

34810 EXAMPLES

34811 None.

34812 APPLICATION USAGE

34813 None.

34814 RATIONALE

34815 Mutex objects are intended to serve as a low-level primitive from which other thread
34816 synchronization functions can be built. As such, the implementation of mutexes should be as
34817 efficient as possible, and this has ramifications on the features available at the interface.

34818 The mutex functions and the particular default settings of the mutex attributes have been
34819 motivated by the desire to not preclude fast, inlined implementations of mutex locking and
34820 unlocking.

34821 For example, on systems not supporting the XSI extended mutex types, deadlocking on a 2
34822 double-lock is explicitly allowed behavior in order to avoid requiring more overhead in the basic 2
34823 mechanism than is absolutely necessary. (More "friendly" mutexes that detect deadlock or that 2
34824 allow multiple locking by the same thread are easily constructed by the user via the other 2
34825 mechanisms provided. For example, *pthread_self()* can be used to record mutex ownership.)
34826 Implementations might also choose to provide such extended features as options via special 2
34827 mutex attributes.

34828 Since most attributes only need to be checked when a thread is going to be blocked, the use of
34829 attributes does not slow the (common) mutex-locking case.

34830	Likewise, while being able to extract the thread ID of the owner of a mutex might be desirable, it would require storing the current thread ID when each mutex is locked, and this could incur unacceptable levels of overhead. Similar arguments apply to a <i>mutex_tryunlock</i> operation.	
34833	For further rationale on the XSI extended mutex types, see the Rationale (Informative) volume of IEEE Std 1003.1-2001.	2
34834		2
34835	FUTURE DIRECTIONS	
34836	None.	
34837	SEE ALSO	
34838	<i>pthread_mutex_destroy()</i> , <i>pthread_mutex_timedlock()</i> , the Base Definitions volume of	
34839	IEEE Std 1003.1-2001, < <i>pthread.h</i> >	
34840	CHANGE HISTORY	
34841	First released in Issue 5. Included for alignment with the POSIX Threads Extension.	
34842	Issue 6	
34843	The <i>pthread_mutex_lock()</i> , <i>pthread_mutex_trylock()</i> , and <i>pthread_mutex_unlock()</i> functions are marked as part of the Threads option.	
34844		
34845	The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:	
34846		
34847	• The behavior when attempting to relock a mutex is defined.	
34848	The <i>pthread_mutex_timedlock()</i> function is added to the SEE ALSO section for alignment with	
34849	IEEE Std 1003.1d-1999.	
34850	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/98 is applied, updating the ERRORS	2
34851	section so that the [EDEADLK] error includes detection of a deadlock condition. The	2
34852	RATIONALE section is also reworded to take into account non-XSI-conformant systems.	2

34853 **NAME**

34854 `pthread_mutex_setprioceiling` — change the priority ceiling of a mutex (**REALTIME THREADS**)

34856 **SYNOPSIS**

34857 THR TPP `#include <pthread.h>`

```
34858     int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,  
34859         int prioceiling, int *restrict old_ceiling);  
34860
```

34861 **DESCRIPTION**

34862 Refer to *pthread_mutex_getprioceiling()*.

34863 NAME

34864 pthread_mutex_timedlock — lock a mutex (ADVANCED REALTIME)

34865 SYNOPSIS

```
34866 THR TMO #include <pthread.h>
34867     #include <time.h>
34868     int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
34869             const struct timespec *restrict abs_timeout);
```

34871 DESCRIPTION

34872 The *pthread_mutex_timedlock()* function shall lock the mutex object referenced by *mutex*. If the
 34873 mutex is already locked, the calling thread shall block until the mutex becomes available as in
 34874 the *pthread_mutex_lock()* function. If the mutex cannot be locked without waiting for another
 34875 thread to unlock the mutex, this wait shall be terminated when the specified timeout expires.

34876 The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by
 34877 the clock on which timeouts are based (that is, when the value of that clock equals or exceeds
 34878 *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time
 34879 of the call.

34880 TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if
 34881 the Timers option is not supported, the timeout shall be based on the system clock as returned
 34882 by the *time()* function.

34883 The resolution of the timeout shall be the resolution of the clock on which it is based. The
 34884 **timespec** data type is defined in the <time.h> header.

34885 Under no circumstance shall the function fail with a timeout if the mutex can be locked
 34886 immediately. The validity of the *abs_timeout* parameter need not be checked if the mutex can be
 34887 locked immediately.

34888 TPI As a consequence of the priority inheritance rules (for mutexes initialized with the
 34889 PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the
 34890 priority of the owner of the mutex shall be adjusted as necessary to reflect the fact that this
 34891 thread is no longer among the threads waiting for the mutex. 2

34892 RETURN VALUE

34893 If successful, the *pthread_mutex_timedlock()* function shall return zero; otherwise, an error
 34894 number shall be returned to indicate the error.

34895 ERRORS

34896 The *pthread_mutex_timedlock()* function shall fail if:

34897 [EINVAL] The mutex was created with the protocol attribute having the value
 34898 PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than
 34899 the mutex' current priority ceiling.

34900 [EINVAL] The process or thread would have blocked, and the *abs_timeout* parameter
 34901 specified a nanoseconds field value less than zero or greater than or equal to
 34902 1 000 million.

34903 [ETIMEDOUT] The mutex could not be locked before the specified timeout expired.

34904 The *pthread_mutex_timedlock()* function may fail if:

34905 [EINVAL] The value specified by *mutex* does not refer to an initialized mutex object.

34906	XSI	[EAGAIN]	The mutex could not be acquired because the maximum number of recursive locks for <i>mutex</i> has been exceeded.	
34907				
34908		[EDEADLK]	A deadlock condition was detected or the current thread already owns the mutex.	2
34909				2
34910			This function shall not return an error code of [EINTR].	
34911	EXAMPLES			
34912			None.	
34913	APPLICATION USAGE			
34914			The <i>pthread_mutex_timedlock()</i> function is part of the Threads and Timeouts options and need	
34915			not be provided on all implementations.	
34916	RATIONALE			
34917			None.	
34918	FUTURE DIRECTIONS			
34919			None.	
34920	SEE ALSO			
34921			<i>pthread_mutex_destroy()</i> , <i>pthread_mutex_lock()</i> , <i>pthread_mutex_trylock()</i> , <i>time()</i> , the Base	
34922			Definitions volume of IEEE Std 1003.1-2001, < <i>pthread.h</i> >, < <i>time.h</i> >	
34923	CHANGE HISTORY			
34924			First released in Issue 6. Derived from IEEE Std 1003.1d-1999.	
34925			IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/99 is applied, marking the last paragraph	2
34926			in the DESCRIPTION as part of the Thread Priority Inheritance option.	2
34927			IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/100 is applied, updating the ERRORS	2
34928			section so that the [EDEADLK] error includes detection of a deadlock condition.	2

34929 **NAME**

34930 pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

34931 **SYNOPSIS**

34932 THR #include <pthread.h>

34933 int pthread_mutex_trylock(pthread_mutex_t *mutex);

34934 int pthread_mutex_unlock(pthread_mutex_t *mutex);

34935

34936 **DESCRIPTION**34937 Refer to *pthread_mutex_lock()*.

34938 NAME

34939 pthread_mutexattr_destroy, pthread_mutexattr_init — destroy and initialize the mutex
 34940 attributes object

34941 SYNOPSIS

```
34942 THR #include <pthread.h>
34943
34944     int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
34945     int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

34946 DESCRIPTION

34947 The *pthread_mutexattr_destroy()* function shall destroy a mutex attributes object; the object
 34948 becomes, in effect, uninitialized. An implementation may cause *pthread_mutexattr_destroy()* to
 34949 set the object referenced by *attr* to an invalid value. A destroyed *attr* attributes object can be
 34950 reinitialized using *pthread_mutexattr_init()*; the results of otherwise referencing the object after it
 34951 has been destroyed are undefined.

34952 The *pthread_mutexattr_init()* function shall initialize a mutex attributes object *attr* with the
 34953 default value for all of the attributes defined by the implementation.

34954 Results are undefined if *pthread_mutexattr_init()* is called specifying an already initialized *attr*
 34955 attributes object.

34956 After a mutex attributes object has been used to initialize one or more mutexes, any function
 34957 affecting the attributes object (including destruction) shall not affect any previously initialized
 34958 mutexes.

34959 RETURN VALUE

34960 Upon successful completion, *pthread_mutexattr_destroy()* and *pthread_mutexattr_init()* shall
 34961 return zero; otherwise, an error number shall be returned to indicate the error.

34962 ERRORS

34963 The *pthread_mutexattr_destroy()* function may fail if:

34964 [EINVAL] The value specified by *attr* is invalid.

34965 The *pthread_mutexattr_init()* function shall fail if:

34966 [ENOMEM] Insufficient memory exists to initialize the mutex attributes object.

34967 These functions shall not return an error code of [EINTR].

34968 EXAMPLES

34969 None.

34970 APPLICATION USAGE

34971 None.

34972 RATIONALE

34973 See *pthread_attr_init()* for a general explanation of attributes. Attributes objects allow
 34974 implementations to experiment with useful extensions and permit extension of this volume of
 34975 IEEE Std 1003.1-2001 without changing the existing functions. Thus, they provide for future
 34976 extensibility of this volume of IEEE Std 1003.1-2001 and reduce the temptation to standardize
 34977 prematurely on semantics that are not yet widely implemented or understood.

34978 Examples of possible additional mutex attributes that have been discussed are *spin_only*,
 34979 *limited_spin*, *no_spin*, *recursive*, and *metered*. (To explain what the latter attributes might mean:
 34980 recursive mutexes would allow for multiple re-locking by the current owner; metered mutexes
 34981 would transparently keep records of queue length, wait time, and so on.) Since there is not yet

34982 wide agreement on the usefulness of these resulting from shared implementation and usage
34983 experience, they are not yet specified in this volume of IEEE Std 1003.1-2001. Mutex attributes
34984 objects, however, make it possible to test out these concepts for possible standardization at a
34985 later time.

34986 Mutex Attributes and Performance

34987 Care has been taken to ensure that the default values of the mutex attributes have been defined
34988 such that mutexes initialized with the defaults have simple enough semantics so that the locking
34989 and unlocking can be done with the equivalent of a test-and-set instruction (plus possibly a few
34990 other basic instructions).

34991 There is at least one implementation method that can be used to reduce the cost of testing at
34992 lock-time if a mutex has non-default attributes. One such method that an implementation can
34993 employ (and this can be made fully transparent to fully conforming POSIX applications) is to
34994 secretly pre-lock any mutexes that are initialized to non-default attributes. Any later attempt to
34995 lock such a mutex causes the implementation to branch to the “slow path” as if the mutex were
34996 unavailable; then, on the slow path, the implementation can do the “real work” to lock a non-
34997 default mutex. The underlying unlock operation is more complicated since the implementation
34998 never really wants to release the pre-lock on this kind of mutex. This illustrates that, depending
34999 on the hardware, there may be certain optimizations that can be used so that whatever mutex
35000 attributes are considered “most frequently used” can be processed most efficiently.

35001 Process Shared Memory and Synchronization

35002 The existence of memory mapping functions in this volume of IEEE Std 1003.1-2001 leads to the
35003 possibility that an application may allocate the synchronization objects from this section in
35004 memory that is accessed by multiple processes (and therefore, by threads of multiple processes).

35005 In order to permit such usage, while at the same time keeping the usual case (that is, usage
35006 within a single process) efficient, a *process-shared* option has been defined.

35007 If an implementation supports the _POSIX_THREAD_PROCESS_SHARED option, then the
35008 *process-shared* attribute can be used to indicate that mutexes or condition variables may be
35009 accessed by threads of multiple processes.

35010 The default setting of PTHREAD_PROCESS_PRIVATE has been chosen for the *process-shared*
35011 attribute so that the most efficient forms of these synchronization objects are created by default.

35012 Synchronization variables that are initialized with the PTHREAD_PROCESS_PRIVATE *process-*
35013 *shared* attribute may only be operated on by threads in the process that initialized them.
35014 Synchronization variables that are initialized with the PTHREAD_PROCESS_SHARED *process-*
35015 *shared* attribute may be operated on by any thread in any process that has access to it. In
35016 particular, these processes may exist beyond the lifetime of the initializing process. For example,
35017 the following code implements a simple counting semaphore in a mapped file that may be used
35018 by many processes.

```
35019 /* sem.h */  
35020 struct semaphore {  
35021     pthread_mutex_t lock;  
35022     pthread_cond_t nonzero;  
35023     unsigned count;  
35024 };  
35025 typedef struct semaphore semaphore_t;  
35026 semaphore_t *semaphore_create(char *semaphore_name);  
35027 semaphore_t *semaphore_open(char *semaphore_name);
```

```
35028     void semaphore_post(semaphore_t *semap);
35029     void semaphore_wait(semaphore_t *semap);
35030     void semaphore_close(semaphore_t *semap);
35031 
35032     /* sem.c */
35033     #include <sys/types.h>
35034     #include <sys/stat.h>
35035     #include <sys/mman.h>
35036     #include <fcntl.h>
35037     #include <pthread.h>
35038     #include "sem.h"
35039 
35040     semaphore_t *
35041     semaphore_create(char *semaphore_name)
35042     {
35043         int fd;
35044         semaphore_t *semap;
35045         pthread_mutexattr_t psharedm;
35046         pthread_condattr_t psharedc;
35047 
35048         fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
35049         if (fd < 0)
35050             return (NULL);
35051         (void) ftruncate(fd, sizeof(semaphore_t));
35052         (void) pthread_mutexattr_init(&psharedm);
35053         (void) pthread_mutexattr_setpshared(&psharedm,
35054             PTHREAD_PROCESS_SHARED);
35055         (void) pthread_condattr_init(&psharedc);
35056         (void) pthread_condattr_setpshared(&psharedc,
35057             PTHREAD_PROCESS_SHARED);
35058         semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
35059             PROT_READ | PROT_WRITE, MAP_SHARED,
35060             fd, 0);
35061         close (fd);
35062         (void) pthread_mutex_init(&semap->lock, &psharedm);
35063         (void) pthread_cond_init(&semap->nonzero, &psharedc);
35064         semap->count = 0;
35065         return (semap);
35066     }
35067 
35068     semaphore_t *
35069     semaphore_open(char *semaphore_name)
35070     {
35071         int fd;
35072         semaphore_t *semap;
35073 
35074         fd = open(semaphore_name, O_RDWR, 0666);
35075         if (fd < 0)
35076             return (NULL);
35077         semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
35078             PROT_READ | PROT_WRITE, MAP_SHARED,
35079             fd, 0);
35080         close (fd);
35081         return (semap);
35082     }
```

```
35078     void
35079     semaphore_post(semaphore_t *semaph)
35080     {
35081         pthread_mutex_lock(&semaph->lock);
35082         if (semaph->count == 0)
35083             pthread_cond_signal(&semaph->nonzero);
35084         semaph->count++;
35085         pthread_mutex_unlock(&semaph->lock);
35086     }
35087
35088     void
35089     semaphore_wait(semaphore_t *semaph)
35090     {
35091         pthread_mutex_lock(&semaph->lock);
35092         while (semaph->count == 0)
35093             pthread_cond_wait(&semaph->nonzero, &semaph->lock);
35094         semaph->count--;
35095         pthread_mutex_unlock(&semaph->lock);
35096     }
35097
35098     void
35099     semaphore_close(semaphore_t *semaph)
35100     {
35101         munmap((void *) semaph, sizeof(semaphore_t));
35102     }
```

35101 The following code is for three separate processes that create, post, and wait on a semaphore in
35102 the file **/tmp/semaphore**. Once the file is created, the post and wait programs increment and
35103 decrement the counting semaphore (waiting and waking as required) even though they did not
35104 initialize the semaphore.

```
35105 /* create.c */
35106 #include "pthread.h"
35107 #include "sem.h"
35108
35109 int
35110 main()
35111 {
35112     semaphore_t *semaph;
35113
35114     semaph = semaphore_create("/tmp/semaphore");
35115     if (semaph == NULL)
35116         exit(1);
35117     semaphore_close(semaph);
35118     return (0);
35119 }
35120
35121 /* post */
35122 #include "pthread.h"
35123 #include "sem.h"
35124
35125 int
35126 main()
35127 {
35128     semaphore_t *semaph;
```

```
35125     semap = semaphore_open( "/tmp/semaphore" );
35126     if (semap == NULL)
35127         exit(1);
35128     semaphore_post(semap);
35129     semaphore_close(semap);
35130     return (0);
35131 }
35132 /* wait */
35133 #include "pthread.h"
35134 #include "sem.h"
35135 int
35136 main()
35137 {
35138     semaphore_t *semap;
35139     semap = semaphore_open( "/tmp/semaphore" );
35140     if (semap == NULL)
35141         exit(1);
35142     semaphore_wait(semap);
35143     semaphore_close(semap);
35144     return (0);
35145 }
```

35146 FUTURE DIRECTIONS

35147 None.

35148 SEE ALSO

35149 *pthread_cond_destroy()*, *pthread_create()*, *pthread_mutex_destroy()*, *pthread_mutexattr_destroy()*, the
35150 Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

35151 CHANGE HISTORY

35152 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35153 Issue 6

35154 The *pthread_mutexattr_destroy()* and *pthread_mutexattr_init()* functions are marked as part of the
35155 Threads option.

35156 IEEE PASC Interpretation 1003.1c #27 is applied, updating the ERRORS section.

35157 NAME

35158 *pthread_mutexattr_getprioceiling*, *pthread_mutexattr_setprioceiling* — get and set the
35159 prioceiling attribute of the mutex attributes object (**REALTIME THREADS**)

35160 SYNOPSIS

35161 THR TPP #include <pthread.h>

```
35162     int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *
35163             restrict attr, int *restrict prioceiling);
35164     int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
35165             int prioceiling);
```

35167 DESCRIPTION

35168 The *pthread_mutexattr_getprioceiling()* and *pthread_mutexattr_setprioceiling()* functions,
35169 respectively, shall get and set the priority ceiling attribute of a mutex attributes object pointed to
35170 by *attr* which was previously created by the function *pthread_mutexattr_init()*.

35171 The *prioceiling* attribute contains the priority ceiling of initialized mutexes. The values of
35172 *prioceiling* are within the maximum range of priorities defined by SCHED_FIFO.

35173 The *prioceiling* attribute defines the priority ceiling of initialized mutexes, which is the minimum
35174 priority level at which the critical section guarded by the mutex is executed. In order to avoid
35175 priority inversion, the priority ceiling of the mutex shall be set to a priority higher than or equal
35176 to the highest priority of all the threads that may lock that mutex. The values of *prioceiling* are
35177 within the maximum range of priorities defined under the SCHED_FIFO scheduling policy.

35178 RETURN VALUE

35179 Upon successful completion, the *pthread_mutexattr_getprioceiling()* and
35180 *pthread_mutexattr_setprioceiling()* functions shall return zero; otherwise, an error number shall be
35181 returned to indicate the error.

35182 ERRORS

35183 The *pthread_mutexattr_getprioceiling()* and *pthread_mutexattr_setprioceiling()* functions may fail if:
35184 [EINVAL] The value specified by *attr* or *prioceiling* is invalid.
35185 [EPERM] The caller does not have the privilege to perform the operation.
35186 These functions shall not return an error code of [EINTR].

35187 EXAMPLES

35188 None.

35189 APPLICATION USAGE

35190 None.

35191 RATIONALE

35192 None.

35193 FUTURE DIRECTIONS

35194 None.

35195 SEE ALSO

35196 *pthread_cond_destroy()*, *pthread_create()*, *pthread_mutex_destroy()*, the Base Definitions volume of
35197 IEEE Std 1003.1-2001, <pthread.h>

35198 CHANGE HISTORY

35199 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35200 Marked as part of the Realtime Threads Feature Group.

35201 Issue 6

35202 The `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions are marked as part of the Threads and Thread Priority Protection options.

35204 The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Priority Protection option.

35206 The [ENOTSUP] error condition has been removed since these functions do not have a *protocol* argument.

35208 The **restrict** keyword is added to the `pthread_mutexattr_getprioceiling()` prototype for alignment with the ISO/IEC 9899:1999 standard.

35210 NAME

35211 pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol — get and set the protocol
 35212 attribute of the mutex attributes object (**REALTIME THREADS**)

35213 SYNOPSIS

```
35214 THR #include <pthread.h>
35215 TPP|TPI int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *
35216           restrict attr, int *restrict protocol);
35217 int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
35218           int protocol);
```

35219

35220 DESCRIPTION

35221 The *pthread_mutexattr_getprotocol()* and *pthread_mutexattr_setprotocol()* functions, respectively,
 35222 shall get and set the protocol attribute of a mutex attributes object pointed to by *attr* which was
 35223 previously created by the function *pthread_mutexattr_init()*.

35224 The *protocol* attribute defines the protocol to be followed in utilizing mutexes. The value of
 35225 *protocol* may be one of:

```
35226 PTHREAD_PRIO_NONE
35227 TPI PTHREAD_PRIO_INHERIT
35228 TPP PTHREAD_PRIO_PROTECT
35229
```

35230 which are defined in the <**pthread.h**> header.

35231 When a thread owns a mutex with the PTHREAD_PRIO_NONE *protocol* attribute, its priority
 35232 and scheduling shall not be affected by its mutex ownership.

35233 TPI When a thread is blocking higher priority threads because of owning one or more mutexes with
 35234 the PTHREAD_PRIO_INHERIT *protocol* attribute, it shall execute at the higher of its priority or
 35235 the priority of the highest priority thread waiting on any of the mutexes owned by this thread
 35236 and initialized with this protocol.

35237 TPP When a thread owns one or more mutexes initialized with the PTHREAD_PRIO_PROTECT
 35238 protocol, it shall execute at the higher of its priority or the highest of the priority ceilings of all
 35239 the mutexes owned by this thread and initialized with this attribute, regardless of whether other
 35240 threads are blocked on any of these mutexes or not.

35241 While a thread is holding a mutex which has been initialized with the
 35242 PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT protocol attributes, it shall not be
 35243 subject to being moved to the tail of the scheduling queue at its priority in the event that its
 35244 original priority is changed, such as by a call to *sched_setparam()*. Likewise, when a thread
 35245 unlocks a mutex that has been initialized with the PTHREAD_PRIO_INHERIT or
 35246 PTHREAD_PRIO_PROTECT protocol attributes, it shall not be subject to being moved to the tail
 35247 of the scheduling queue at its priority in the event that its original priority is changed.

35248 If a thread simultaneously owns several mutexes initialized with different protocols, it shall
 35249 execute at the highest of the priorities that it would have obtained by each of these protocols.

35250 TPI When a thread makes a call to *pthread_mutex_lock()*, the mutex was initialized with the protocol
 35251 attribute having the value PTHREAD_PRIO_INHERIT, when the calling thread is blocked
 35252 because the mutex is owned by another thread, that owner thread shall inherit the priority level
 35253 of the calling thread as long as it continues to own the mutex. The implementation shall update
 35254 its execution priority to the maximum of its assigned priority and all its inherited priorities.
 35255 Furthermore, if this owner thread itself becomes blocked on another mutex, the same priority

35256 inheritance effect shall be propagated to this other owner thread, in a recursive manner.

35257 RETURN VALUE

35258 Upon successful completion, the *pthread_mutexattr_getprotocol()* and
35259 *pthread_mutexattr_setprotocol()* functions shall return zero; otherwise, an error number shall be
35260 returned to indicate the error.

35261 ERRORS

35262 The *pthread_mutexattr_setprotocol()* function shall fail if:

35263 [ENOTSUP] The value specified by *protocol* is an unsupported value.

35264 The *pthread_mutexattr_getprotocol()* and *pthread_mutexattr_setprotocol()* functions may fail if:

35265 [EINVAL] The value specified by *attr* or *protocol* is invalid.

35266 [EPERM] The caller does not have the privilege to perform the operation.

35267 These functions shall not return an error code of [EINTR].

35268 EXAMPLES

35269 None.

35270 APPLICATION USAGE

35271 None.

35272 RATIONALE

35273 None.

35274 FUTURE DIRECTIONS

35275 None.

35276 SEE ALSO

35277 *pthread_cond_destroy()*, *pthread_create()*, *pthread_mutex_destroy()*, the Base Definitions volume of
35278 IEEE Std 1003.1-2001, <*pthread.h*>

35279 CHANGE HISTORY

35280 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35281 Marked as part of the Realtime Threads Feature Group.

35282 Issue 6

35283 The *pthread_mutexattr_getprotocol()* and *pthread_mutexattr_setprotocol()* functions are marked as
35284 part of the Threads option and either the Thread Priority Protection or Thread Priority
35285 Inheritance options.

35286 The [ENOSYS] error condition has been removed as stubs need not be provided if an
35287 implementation does not support the Thread Priority Protection or Thread Priority Inheritance
35288 options.

35289 The **restrict** keyword is added to the *pthread_mutexattr_getprotocol()* prototype for alignment
35290 with the ISO/IEC 9899:1999 standard.

35291 NAME

35292 `pthread_mutexattr_getpshared`, `pthread_mutexattr_setpshared` — get and set the process-
35293 shared attribute

35294 SYNOPSIS

35295 THR TSH `#include <pthread.h>`

```
35296     int pthread_mutexattr_getpshared(const pthread_mutexattr_t *
35297             restrict attr, int *restrict pshared);
35298     int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
35299             int pshared);
```

35300

35301 DESCRIPTION

35302 The `pthread_mutexattr_getpshared()` function shall obtain the value of the *process-shared* attribute
35303 from the attributes object referenced by `attr`. The `pthread_mutexattr_setpshared()` function shall
35304 set the *process-shared* attribute in an initialized attributes object referenced by `attr`.

35305 The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a mutex to be
35306 operated upon by any thread that has access to the memory where the mutex is allocated, even if
35307 the mutex is allocated in memory that is shared by multiple processes. If the *process-shared*
35308 attribute is PTHREAD_PROCESS_PRIVATE, the mutex shall only be operated upon by threads
35309 created within the same process as the thread that initialized the mutex; if threads of differing
35310 processes attempt to operate on such a mutex, the behavior is undefined. The default value of
35311 the attribute shall be PTHREAD_PROCESS_PRIVATE.

35312 RETURN VALUE

35313 Upon successful completion, `pthread_mutexattr_setpshared()` shall return zero; otherwise, an error
35314 number shall be returned to indicate the error.

35315 Upon successful completion, `pthread_mutexattr_getpshared()` shall return zero and store the value
35316 of the *process-shared* attribute of `attr` into the object referenced by the `pshared` parameter.
35317 Otherwise, an error number shall be returned to indicate the error.

35318 ERRORS

35319 The `pthread_mutexattr_getpshared()` and `pthread_mutexattr_setpshared()` functions may fail if:
35320 [EINVAL] The value specified by `attr` is invalid.
35321 The `pthread_mutexattr_setpshared()` function may fail if:
35322 [EINVAL] The new value specified for the attribute is outside the range of legal values
35323 for that attribute.
35324 These functions shall not return an error code of [EINTR].

35325 EXAMPLES

35326 None.

35327 APPLICATION USAGE

35328 None.

35329 RATIONALE

35330 None.

35331 FUTURE DIRECTIONS

35332 None.

35333 **SEE ALSO**

35334 *pthread_cond_destroy()*, *pthread_create()*, *pthread_mutex_destroy()*, *pthread_mutexattr_destroy()*, the
35335 Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

35336 **CHANGE HISTORY**

35337 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35338 **Issue 6**

35339 The *pthread_mutexattr_getpshared()* and *pthread_mutexattr_setpshared()* functions are marked as
35340 part of the Threads and Thread Process-Shared Synchronization options.

35341 The **restrict** keyword is added to the *pthread_mutexattr_getpshared()* prototype for alignment
35342 with the ISO/IEC 9899:1999 standard.

35343 NAME

35344 pthread_mutexattr_gettype, pthread_mutexattr_settype — get and set the mutex type attribute

35345 SYNOPSIS

35346 XSI #include <pthread.h>

```
35347 int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
35348     int *restrict type);
35349 int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

35351 DESCRIPTION

35352 The *pthread_mutexattr_gettype()* and *pthread_mutexattr_settype()* functions, respectively, shall get
35353 and set the mutex *type* attribute. This attribute is set in the *type* parameter to these functions. The
35354 default value of the *type* attribute is PTHREAD_MUTEX_DEFAULT.

35355 The type of mutex is contained in the *type* attribute of the mutex attributes. Valid mutex types
35356 include:

35357 PTHREAD_MUTEX_NORMAL

35358 This type of mutex does not detect deadlock. A thread attempting to relock this mutex
35359 without first unlocking it shall deadlock. Attempting to unlock a mutex locked by a
35360 different thread results in undefined behavior. Attempting to unlock an unlocked mutex
35361 results in undefined behavior.

35362 PTHREAD_MUTEX_ERRORCHECK

35363 This type of mutex provides error checking. A thread attempting to relock this mutex
35364 without first unlocking it shall return with an error. A thread attempting to unlock a mutex
35365 which another thread has locked shall return with an error. A thread attempting to unlock
35366 an unlocked mutex shall return with an error.

35367 PTHREAD_MUTEX_RECURSIVE

35368 A thread attempting to relock this mutex without first unlocking it shall succeed in locking
35369 the mutex. The relocking deadlock which can occur with mutexes of type
35370 PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this
35371 mutex shall require the same number of unlocks to release the mutex before another thread
35372 can acquire the mutex. A thread attempting to unlock a mutex which another thread has
35373 locked shall return with an error. A thread attempting to unlock an unlocked mutex shall
35374 return with an error.

35375 PTHREAD_MUTEX_DEFAULT

35376 Attempting to recursively lock a mutex of this type results in undefined behavior.
35377 Attempting to unlock a mutex of this type which was not locked by the calling thread
35378 results in undefined behavior. Attempting to unlock a mutex of this type which is not
35379 locked results in undefined behavior. An implementation may map this mutex to one of the
35380 other mutex types.

35381 RETURN VALUE

35382 Upon successful completion, the *pthread_mutexattr_gettype()* function shall return zero and store
35383 the value of the *type* attribute of *attr* into the object referenced by the *type* parameter. Otherwise,
35384 an error shall be returned to indicate the error.

35385 If successful, the *pthread_mutexattr_settype()* function shall return zero; otherwise, an error
35386 number shall be returned to indicate the error.

35387 ERRORS

35388 The *pthread_mutexattr_settype()* function shall fail if:

35389 [EINVAL] The value *type* is invalid.

35390 The *pthread_mutexattr_gettype()* and *pthread_mutexattr_settype()* functions may fail if:

35391 [EINVAL] The value specified by *attr* is invalid.

35392 These functions shall not return an error code of [EINTR].

35393 EXAMPLES

35394 None.

35395 APPLICATION USAGE

35396 It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with
35397 condition variables because the implicit unlock performed for a *pthread_cond_timedwait()* or
35398 *pthread_cond_wait()* may not actually release the mutex (if it had been locked multiple times). If
35399 this happens, no other thread can satisfy the condition of the predicate.

35400 RATIONALE

35401 None.

35402 FUTURE DIRECTIONS

35403 None.

35404 SEE ALSO

35405 *pthread_cond_timedwait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

35406 CHANGE HISTORY

35407 First released in Issue 5.

35408 Issue 6

35409 The Open Group Corrigendum U033/3 is applied. The SYNOPSIS for
35410 *pthread_mutexattr_gettype()* is updated so that the first argument is of type **const**
35411 **pthread_mutexattr_t** *.

35412 The **restrict** keyword is added to the *pthread_mutexattr_gettype()* prototype for alignment with
35413 the ISO/IEC 9899:1999 standard.

35414 NAME

35415 pthread_mutexattr_init — initialize the mutex attributes object

35416 SYNOPSIS

35417 THR #include <pthread.h>

35418 int pthread_mutexattr_init(pthread_mutexattr_t *attr);

35419

35420 DESCRIPTION

35421 Refer to *pthread_mutexattr_destroy()*.

35422 NAME

35423 **pthread_mutexattr_setprioceiling** — set the prioceiling attribute of the mutex attributes object
35424 **(REALTIME THREADS)**

35425 SYNOPSIS

35426 THR TPP #include <pthread.h>
35427 int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
35428 int prioceiling);
35429

35430 DESCRIPTION

35431 Refer to *pthread_mutexattr_getprioceiling()*.

35432 NAME

35433 **pthread_mutexattr_setprotocol** — set the protocol attribute of the mutex attributes object
35434 **(REALTIME THREADS)**

35435 SYNOPSIS

35436 THR `#include <pthread.h>`

35437 TPP|TPI `int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,`
35438 `int protocol);`

35439

35440 DESCRIPTION

35441 Refer to *pthread_mutexattr_getprotocol()*.

35442 NAME

35443 `pthread_mutexattr_setpshared` — set the process-shared attribute

35444 SYNOPSIS

35445 THR TSH `#include <pthread.h>`

35446 `int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,`

35447 `int pshared);`

35448

35449 DESCRIPTION

35450 Refer to `pthread_mutexattr_getpshared()`.

35451 **NAME**

35452 pthread_mutexattr_settype — set the mutex type attribute

35453 **SYNOPSIS**

35454 XSI #include <pthread.h>

35455 int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);

35456

35457 **DESCRIPTION**

35458 Refer to *pthread_mutexattr_gettype()*.

35459 NAME

35460 *pthread_once* — dynamic package initialization

35461 SYNOPSIS

```
35462 THR #include <pthread.h>
35463     int pthread_once(pthread_once_t *once_control,
35464             void (*init_routine)(void));
35465     pthread_once_t once_control = PTHREAD_ONCE_INIT;
35466
```

35467 DESCRIPTION

35468 The first call to *pthread_once()* by any thread in a process, with a given *once_control*, shall call the
35469 *init_routine* with no arguments. Subsequent calls of *pthread_once()* with the same *once_control*
35470 shall not call the *init_routine*. On return from *pthread_once()*, *init_routine* shall have completed.
35471 The *once_control* parameter shall determine whether the associated initialization routine has
35472 been called.

35473 The *pthread_once()* function is not a cancellation point. However, if *init_routine* is a cancellation
35474 point and is canceled, the effect on *once_control* shall be as if *pthread_once()* was never called.

35475 The constant PTHREAD_ONCE_INIT is defined in the <**pthread.h**> header.

35476 The behavior of *pthread_once()* is undefined if *once_control* has automatic storage duration or is
35477 not initialized by PTHREAD_ONCE_INIT.

35478 RETURN VALUE

35479 Upon successful completion, *pthread_once()* shall return zero; otherwise, an error number shall
35480 be returned to indicate the error.

35481 ERRORS

35482 The *pthread_once()* function may fail if:

35483 [EINVAL] If either *once_control* or *init_routine* is invalid.

35484 The *pthread_once()* function shall not return an error code of [EINTR].

35485 EXAMPLES

35486 None.

35487 APPLICATION USAGE

35488 None.

35489 RATIONALE

35490 Some C libraries are designed for dynamic initialization. That is, the global initialization for the
35491 library is performed when the first procedure in the library is called. In a single-threaded
35492 program, this is normally implemented using a static variable whose value is checked on entry
35493 to a routine, as follows:

```
35494     static int random_is_initialized = 0;
35495     extern int initialize_random();
35496
35497     int random_function()
35498     {
35499         if (random_is_initialized == 0) {
35500             initialize_random();
35501             random_is_initialized = 1;
35502         }
35503         /* Operations performed after initialization. */
35504     }
```

35504 To keep the same structure in a multi-threaded program, a new primitive is needed. Otherwise,
35505 library initialization has to be accomplished by an explicit call to a library-exported initialization
35506 function prior to any use of the library.

35507 For dynamic library initialization in a multi-threaded process, a simple initialization flag is not
35508 sufficient; the flag needs to be protected against modification by multiple threads
35509 simultaneously calling into the library. Protecting the flag requires the use of a mutex; however,
35510 mutexes have to be initialized before they are used. Ensuring that the mutex is only initialized
35511 once requires a recursive solution to this problem.

35512 The use of *pthread_once()* not only supplies an implementation-guaranteed means of dynamic
35513 initialization, it provides an aid to the reliable construction of multi-threaded and realtime
35514 systems. The preceding example then becomes:

```
35515 #include <pthread.h>
35516 static pthread_once_t random_is_initialized = PTHREAD_ONCE_INIT;
35517 extern int initialize_random();
35518
35519     int random_function()
35520     {
35521         (void) pthread_once(&random_is_initialized, initialize_random);
35522         /* Operations performed after initialization. */
35523     }
```

35523 Note that a **pthread_once_t** cannot be an array because some compilers do not accept the
35524 construct **&<array_name>**.

35525 FUTURE DIRECTIONS

35526 None.

35527 SEE ALSO

35528 The Base Definitions volume of IEEE Std 1003.1-2001, **<pthread.h>**

35529 CHANGE HISTORY

35530 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

35531 Issue 6

35532 The *pthread_once()* function is marked as part of the Threads option.

35533 The [EINVAL] error is added as a may fail case for if either argument is invalid.

35534 NAME

35535 *pthread_rwlock_destroy*, *pthread_rwlock_init* — destroy and initialize a read-write lock object

35536 SYNOPSIS

35537 THR *#include <pthread.h>*

```
35538     int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
35539     int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
35540                 const pthread_rwlockattr_t *restrict attr);
```

35542 DESCRIPTION

35543 The *pthread_rwlock_destroy()* function shall destroy the read-write lock object referenced by *rwlock* and release any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is reinitialized by another call to *pthread_rwlock_init()*. An implementation may cause *pthread_rwlock_destroy()* to set the object referenced by *rwlock* to an invalid value. Results are undefined if *pthread_rwlock_destroy()* is called when any thread holds *rwlock*. Attempting to destroy an uninitialized read-write lock results in undefined behavior.

35549 The *pthread_rwlock_init()* function shall allocate any resources required to use the read-write lock referenced by *rwlock* and initializes the lock to an unlocked state with attributes referenced by *attr*. If *attr* is NULL, the default read-write lock attributes shall be used; the effect is the same as passing the address of a default read-write lock attributes object. Once initialized, the lock can be used any number of times without being reinitialized. Results are undefined if *pthread_rwlock_init()* is called specifying an already initialized read-write lock. Results are undefined if a read-write lock is used without first being initialized.

35556 If the *pthread_rwlock_init()* function fails, *rwlock* shall not be initialized and the contents of *rwlock* are undefined.

35558 Only the object referenced by *rwlock* may be used for performing synchronization. The result of referring to copies of that object in calls to *pthread_rwlock_destroy()*, *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*, *pthread_rwlock_timedwrlock()*, *pthread_rwlock_tryrdlock()*, *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*, or *pthread_rwlock_wrlock()* is undefined.

35562 RETURN VALUE

35563 If successful, the *pthread_rwlock_destroy()* and *pthread_rwlock_init()* functions shall return zero; otherwise, an error number shall be returned to indicate the error.

35565 The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the read-write lock specified by *rwlock*.

35568 ERRORS

35569 The *pthread_rwlock_destroy()* function may fail if:

35570 [EBUSY] The implementation has detected an attempt to destroy the object referenced by *rwlock* while it is locked.

35572 [EINVAL] The value specified by *rwlock* is invalid.

35573 The *pthread_rwlock_init()* function shall fail if:

35574 [EAGAIN] The system lacked the necessary resources (other than memory) to initialize another read-write lock.

35576 [ENOMEM] Insufficient memory exists to initialize the read-write lock.

35577 [EPERM] The caller does not have the privilege to perform the operation.

35578 The *pthread_rwlock_init()* function may fail if:

35579 [EBUSY] The implementation has detected an attempt to reinitialize the object
35580 referenced by *rwlock*, a previously initialized but not yet destroyed read-write
35581 lock.

35582 [EINVAL] The value specified by *attr* is invalid.

35583 These functions shall not return an error code of [EINTR].

35584 EXAMPLES

35585 None.

35586 APPLICATION USAGE

35587 Applications using these and related read-write lock functions may be subject to priority 1
35588 inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.285, 1
35589 Priority Inversion.

35590 RATIONALE

35591 None.

35592 FUTURE DIRECTIONS

35593 None.

35594 SEE ALSO

35595 *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*, *pthread_rwlock_timedwrlock()*,
35596 *pthread_rwlock_tryrdlock()*, *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*,
35597 *pthread_rwlock_wrlock()*, the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

35598 CHANGE HISTORY

35599 First released in Issue 5.

35600 Issue 6

35601 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 35602 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is
35603 now part of the Threads option (previously it was part of the Read-Write Locks option in
35604 IEEE Std 1003.1j-2000 and also part of the XSI extension). The initializer macro is also deleted
35605 from the SYNOPSIS.
- 35606 • The DESCRIPTION is updated as follows:
 - 35607 — It explicitly notes allocation of resources upon initialization of a read-write lock object.
 - 35608 — A paragraph is added specifying that copies of read-write lock objects may not be used.
- 35609 • An [EINVAL] error is added to the ERRORS section for *pthread_rwlock_init()*, indicating that
35610 the *rwlock* value is invalid.
- 35611 • The SEE ALSO section is updated.

35612 The **restrict** keyword is added to the *pthread_rwlock_init()* prototype for alignment with the
35613 ISO/IEC 9899:1999 standard.

35614 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/45 is applied, adding APPLICATION 1
35615 USAGE relating to priority inversion.

35616 NAME

35617 **pthread_rwlock_rdlock**, **pthread_rwlock_tryrdlock** — lock a read-write lock object for reading

35618 SYNOPSIS

35619 **THR** `#include <pthread.h>`

35620 `int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);`

35621 `int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);`

35622

35623 DESCRIPTION

35624 The **pthread_rwlock_rdlock()** function shall apply a read lock to the read-write lock referenced by *rwlock*. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock.

35627 **TPS** If the Thread Execution Scheduling option is supported, and the threads involved in the lock are executing with the scheduling policies **SCHED_FIFO** or **SCHED_RR**, the calling thread shall not acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on the lock; otherwise, the calling thread shall acquire the lock.

35631 **TPS TSP** If the Threads Execution Scheduling option is supported, and the threads involved in the lock are executing with the **SCHED_SPORADIC** scheduling policy, the calling thread shall not acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on the lock; otherwise, the calling thread shall acquire the lock.

35635 If the Thread Execution Scheduling option is not supported, it is implementation-defined whether the calling thread acquires the lock when a writer does not hold the lock and there are writers blocked on the lock. If a writer holds the lock, the calling thread shall not acquire the read lock. If the read lock is not acquired, the calling thread shall block until it can acquire the lock. The calling thread may deadlock if at the time the call is made it holds a write lock.

35640 A thread may hold multiple concurrent read locks on *rwlock* (that is, successfully call the **pthread_rwlock_rdlock()** function *n* times). If so, the application shall ensure that the thread performs matching unlocks (that is, it calls the **pthread_rwlock_unlock()** function *n* times).

35643 The maximum number of simultaneous read locks that an implementation guarantees can be applied to a read-write lock shall be implementation-defined. The **pthread_rwlock_rdlock()** function may fail if this maximum would be exceeded.

35646 The **pthread_rwlock_tryrdlock()** function shall apply a read lock as in the **pthread_rwlock_rdlock()** function, with the exception that the function shall fail if the equivalent **pthread_rwlock_rdlock()** call would have blocked the calling thread. In no case shall the **pthread_rwlock_tryrdlock()** function ever block; it always either acquires the lock or fails and returns immediately.

35650 Results are undefined if any of these functions are called with an uninitialized read-write lock.

35651 If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the signal handler the thread resumes waiting for the read-write lock for reading as if it was not interrupted.

35654 RETURN VALUE

35655 If successful, the **pthread_rwlock_rdlock()** function shall return zero; otherwise, an error number shall be returned to indicate the error.

35657 The **pthread_rwlock_tryrdlock()** function shall return zero if the lock for reading on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to indicate the error.

35660 ERRORS

35661 The *pthread_rwlock_tryrdlock()* function shall fail if:

35662 [EBUSY] The read-write lock could not be acquired for reading because a writer holds the lock or a writer with the appropriate priority was blocked on it.

35664 The *pthread_rwlock_rdlock()* and *pthread_rwlock_tryrdlock()* functions may fail if:

35665 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock object.

35667 [EAGAIN] The read lock could not be acquired because the maximum number of read locks for *rwlock* has been exceeded.

35669 The *pthread_rwlock_rdlock()* function may fail if:

35670 [EDEADLK] A deadlock condition was detected or the current thread already owns the read-write lock for writing. 2

35672 These functions shall not return an error code of [EINTR].

35673 EXAMPLES

35674 None.

35675 APPLICATION USAGE

35676 Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

35678 RATIONALE

35679 None.

35680 FUTURE DIRECTIONS

35681 None.

35682 SEE ALSO

35683 *pthread_rwlock_destroy()*, *pthread_rwlock_timedrdlock()*, *pthread_rwlock_timedwrlock()*,
35684 *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*, *pthread_rwlock_wrlock()*, the Base Definitions
35685 volume of IEEE Std 1003.1-2001, <pthread.h>

35686 CHANGE HISTORY

35687 First released in Issue 5.

35688 Issue 6

35689 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 35690 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- 35693 • The DESCRIPTION is updated as follows:
 - 35694 — Conditions under which writers have precedence over readers are specified.
 - 35695 — Failure of *pthread_rwlock_tryrdlock()* is clarified.
 - 35696 — A paragraph on the maximum number of read locks is added.
- 35697 • In the ERRORS sections, [EBUSY] is modified to take into account write priority, and [EDEADLK] is deleted as a *pthread_rwlock_tryrdlock()* error.
- 35699 • The SEE ALSO section is updated.

35700
35701IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/101 is applied, updating the ERRORS 2
section so that the [EDEADLK] error includes detection of a deadlock condition. 2

35702 NAME

35703 pthread_rwlock_timedrdlock — lock a read-write lock for reading

35704 SYNOPSIS

```
35705 THR TMO #include <pthread.h>
35706     #include <time.h>
35707
35708     int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
35709             const struct timespec *restrict abs_timeout);
```

35710 DESCRIPTION

35711 The *pthread_rwlock_timedrdlock()* function shall apply a read lock to the read-write lock
 35712 referenced by *rwlock* as in the *pthread_rwlock_rdlock()* function. However, if the lock cannot be
 35713 acquired without waiting for other threads to unlock the lock, this wait shall be terminated
 35714 when the specified timeout expires. The timeout shall expire when the absolute time specified
 35715 by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the
 35716 value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout*
 35717 has already been passed at the time of the call.

35718 TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock. If
 35719 the Timers option is not supported, the timeout shall be based on the system clock as returned
 35720 by the *time()* function. The resolution of the timeout shall be the resolution of the clock on which
 35721 it is based. The *timespec* data type is defined in the <time.h> header. Under no circumstances
 35722 shall the function fail with a timeout if the lock can be acquired immediately. The validity of the
 35723 *abs_timeout* parameter need not be checked if the lock can be immediately acquired.

35724 If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-
 35725 write lock via a call to *pthread_rwlock_timedrdlock()*, upon return from the signal handler the
 35726 thread shall resume waiting for the lock as if it was not interrupted.

35727 The calling thread may deadlock if at the time the call is made it holds a write lock on *rwlock*.
 35728 The results are undefined if this function is called with an uninitialized read-write lock.

35729 RETURN VALUE

35730 The *pthread_rwlock_timedrdlock()* function shall return zero if the lock for reading on the read-
 35731 write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned
 35732 to indicate the error.

35733 ERRORS

35734 The *pthread_rwlock_timedrdlock()* function shall fail if:

35735 [ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

35736 The *pthread_rwlock_timedrdlock()* function may fail if:

35737 [EAGAIN] The read lock could not be acquired because the maximum number of read
 35738 locks for lock would be exceeded.

35739 [EDEADLK] A deadlock condition was detected or the calling thread already holds a write
 35740 lock on *rwlock*.

35741 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
 35742 object, or the *abs_timeout* nanosecond value is less than zero or greater than or
 35743 equal to 1 000 million.

35744 This function shall not return an error code of [EINTR].

2
2

35745 EXAMPLES

35746 None.

35747 APPLICATION USAGE

35748 Applications using this function may be subject to priority inversion, as discussed in the Base
35749 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

35750 The *pthread_rwlock_timedrdlock()* function is part of the Threads and Timeouts options and need
35751 not be provided on all implementations.

35752 RATIONALE

35753 None.

35754 FUTURE DIRECTIONS

35755 None.

35756 SEE ALSO

35757 *pthread_rwlock_destroy()*, *pthread_rwlock_rdlock()*, *pthread_rwlock_timedwrlock()*,
35758 *pthread_rwlock_tryrdlock()*, *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*,
35759 *pthread_rwlock_wrlock()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>,
35760 <**time.h**>

35761 CHANGE HISTORY

35762 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

35763 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/102 is applied, updating the ERRORS 2
35764 section so that the [EDEADLK] error includes detection of a deadlock condition. 2

35765 NAME

35766 pthread_rwlock_timedwrlock — lock a read-write lock for writing

35767 SYNOPSIS

```
35768 THR TMO #include <pthread.h>
35769     #include <time.h>
35770
35771     int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
35772             const struct timespec *restrict abs_timeout);
```

35773 DESCRIPTION

35774 The *pthread_rwlock_timedwrlock()* function shall apply a write lock to the read-write lock referenced by *rwlock* as in the *pthread_rwlock_wrlock()* function. However, if the lock cannot be acquired without waiting for other threads to unlock the lock, this wait shall be terminated when the specified timeout expires. 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.

35781 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 in the <time.h> header. Under no circumstances shall the function fail with a timeout if the lock can be acquired immediately. The validity of the *abs_timeout* parameter need not be checked if the lock can be immediately acquired.

35787 If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-write lock via a call to *pthread_rwlock_timedwrlock()*, upon return from the signal handler the thread shall resume waiting for the lock as if it was not interrupted.

35790 The calling thread may deadlock if at the time the call is made it holds the read-write lock. The results are undefined if this function is called with an uninitialized read-write lock.

35792 RETURN VALUE

35793 The *pthread_rwlock_timedwrlock()* function shall return zero if the lock for writing on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to indicate the error.

35796 ERRORS

35797 The *pthread_rwlock_timedwrlock()* function shall fail if:

35798 [ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

35799 The *pthread_rwlock_timedwrlock()* function may fail if:

35800 [EDEADLK] A deadlock condition was detected or the calling thread already holds the 2
35801 *rwlock*.

35802 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
35803 object, or the *abs_timeout* nanosecond value is less than zero or greater than or
35804 equal to 1 000 million.

35805 This function shall not return an error code of [EINTR].

35806 EXAMPLES

35807 None.

35808 APPLICATION USAGE

35809 Applications using this function may be subject to priority inversion, as discussed in the Base
35810 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

35811 The *pthread_rwlock_timedwrlock()* function is part of the Threads and Timeouts options and need
35812 not be provided on all implementations.

35813 RATIONALE

35814 None.

35815 FUTURE DIRECTIONS

35816 None.

35817 SEE ALSO

35818 *pthread_rwlock_destroy()*, *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*,
35819 *pthread_rwlock_tryrdlock()*, *pthread_rwlock_trywrlock()*, *pthread_rwlock_unlock()*,
35820 *pthread_rwlock_wrlock()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>,
35821 <**time.h**>

35822 CHANGE HISTORY

35823 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

35824 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/103 is applied, updating the ERRORS 2
35825 section so that the [EDEADLK] error includes detection of a deadlock condition. 2

35826 NAME

35827 `pthread_rwlock_tryrdlock` — lock a read-write lock object for reading

35828 SYNOPSIS

35829 `THR #include <pthread.h>`

35830 `int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);`

35831

35832 DESCRIPTION

35833 Refer to *pthread_rwlock_rdlock()*.

35834 NAME

35835 `pthread_rwlock_trywrlock`, `pthread_rwlock_wrlock` — lock a read-write lock object for writing

35836 SYNOPSIS

35837 THR `#include <pthread.h>`35838 `int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);`35839 `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`

35840

35841 DESCRIPTION

35842 The `pthread_rwlock_trywrlock()` function shall apply a write lock like the `pthread_rwlock_wrlock()` function, with the exception that the function shall fail if any thread currently holds `rwlock` (for reading or writing).

35843 The `pthread_rwlock_wrlock()` function shall apply a write lock to the read-write lock referenced by `rwlock`. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock `rwlock`. Otherwise, the thread shall block until it can acquire the lock. The calling thread may deadlock if at the time the call is made it holds the read-write lock (whether a read or write lock).

35844 Implementations may favor writers over readers to avoid writer starvation.

35845 Results are undefined if any of these functions are called with an uninitialized read-write lock.

35846 If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the signal handler the thread resumes waiting for the read-write lock for writing as if it was not interrupted.

35847 RETURN VALUE

35848 The `pthread_rwlock_trywrlock()` function shall return zero if the lock for writing on the read-write lock object referenced by `rwlock` is acquired. Otherwise, an error number shall be returned to indicate the error.

35849 If successful, the `pthread_rwlock_wrlock()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

35850 ERRORS

35851 The `pthread_rwlock_trywrlock()` function shall fail if:

35852 [EBUSY] The read-write lock could not be acquired for writing because it was already locked for reading or writing.

35853 The `pthread_rwlock_trywrlock()` and `pthread_rwlock_wrlock()` functions may fail if:

35854 [EINVAL] The value specified by `rwlock` does not refer to an initialized read-write lock object.

35855 The `pthread_rwlock_wrlock()` function may fail if:

35856 [EDEADLK] A deadlock condition was detected or the current thread already owns the read-write lock for writing or reading. 2 2

35857 These functions shall not return an error code of [EINTR].

35872 EXAMPLES

35873 None.

35874 APPLICATION USAGE

35875 Applications using these functions may be subject to priority inversion, as discussed in the Base
35876 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

35877 RATIONALE

35878 None.

35879 FUTURE DIRECTIONS

35880 None.

35881 SEE ALSO

35882 *pthread_rwlock_destroy()*, *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*,
35883 *pthread_rwlock_timedwrlock()*, *pthread_rwlock_tryrdlock()*, *pthread_rwlock_unlock()*, the Base
35884 Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

35885 CHANGE HISTORY

35886 First released in Issue 5.

35887 Issue 6

35888 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 35889 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is
35890 now part of the Threads option (previously it was part of the Read-Write Locks option in
35891 IEEE Std 1003.1j-2000 and also part of the XSI extension).
- 35892 • The [EDEADLK] error is deleted as a *pthread_rwlock_trywrlock()* error.
- 35893 • The SEE ALSO section is updated.

35894 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/104 is applied, updating the ERRORS 2
35895 section so that the [EDEADLK] error includes detection of a deadlock condition. 2

35896 NAME

35897 **pthread_rwlock_unlock** — unlock a read-write lock object

35898 SYNOPSIS

35899 **THR** **#include <pthread.h>**

35900 **int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);**

35901

35902 DESCRIPTION

35903 The *pthread_rwlock_unlock()* function shall release a lock held on the read-write lock object referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the calling thread.

35906 If this function is called to release a read lock from the read-write lock object and there are other
35907 read locks currently held on this read-write lock object, the read-write lock object remains in the
35908 read locked state. If this function releases the last read lock for this read-write lock object, the
35909 read-write lock object shall be put in the unlocked state with no owners.

35910 If this function is called to release a write lock for this read-write lock object, the read-write lock
35911 object shall be put in the unlocked state.

35912 If there are threads blocked on the lock when it becomes available, the scheduling policy shall
35913 determine which thread(s) shall acquire the lock. If the Thread Execution Scheduling option is
35914 supported, when threads executing with the scheduling policies SCED_FIFO, SCED_RR, or
35915 SCED_SPORADIC are waiting on the lock, they shall acquire the lock in priority order when
35916 the lock becomes available. For equal priority threads, write locks shall take precedence over
35917 read locks. If the Thread Execution Scheduling option is not supported, it is implementation-
35918 defined whether write locks take precedence over read locks.

35919 Results are undefined if any of these functions are called with an uninitialized read-write lock.

35920 RETURN VALUE

35921 If successful, the *pthread_rwlock_unlock()* function shall return zero; otherwise, an error number
35922 shall be returned to indicate the error.

35923 ERRORS

35924 The *pthread_rwlock_unlock()* function may fail if:

35925 [EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock
35926 object.

35927 [EPERM] The current thread does not hold a lock on the read-write lock.

35928 The *pthread_rwlock_unlock()* function shall not return an error code of [EINTR].

35929 EXAMPLES

35930 None.

35931 APPLICATION USAGE

35932 None.

35933 RATIONALE

35934 None.

35935 FUTURE DIRECTIONS

35936 None.

35937 SEE ALSO

35938 *pthread_rwlock_destroy()*, *pthread_rwlock_rdlock()*, *pthread_rwlock_timedrdlock()*,
35939 *pthread_rwlock_timedwrlock()*, *pthread_rwlock_tryrdlock()*, *pthread_rwlock_trywrlock()*,
35940 *pthread_rwlock_wrlock()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

35941 CHANGE HISTORY

35942 First released in Issue 5.

35943 Issue 6

35944 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 35945 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is
35946 now part of the Threads option (previously it was part of the Read-Write Locks option in
35947 IEEE Std 1003.1j-2000 and also part of the XSI extension).
- 35948 • The DESCRIPTION is updated as follows:
 - 35949 — The conditions under which writers have precedence over readers are specified.
 - 35950 — The concept of read-write lock owner is deleted.
- 35951 • The SEE ALSO section is updated.

35952 NAME

35953 `pthread_rwlock_wrlock` — lock a read-write lock object for writing

35954 SYNOPSIS

35955 `THR #include <pthread.h>`

35956 `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`

35957

35958 DESCRIPTION

35959 Refer to *pthread_rwlock_trywrlock()*.

35960 NAME

35961 *pthread_rwlockattr_destroy*, *pthread_rwlockattr_init* — destroy and initialize the read-write
35962 lock attributes object

35963 SYNOPSIS

35964 THR #include <pthread.h>

35965 int *pthread_rwlockattr_destroy*(*pthread_rwlockattr_t* **attr*);
35966 int *pthread_rwlockattr_init*(*pthread_rwlockattr_t* **attr*);
35967

35968 DESCRIPTION

35969 The *pthread_rwlockattr_destroy()* function shall destroy a read-write lock attributes object. A
35970 destroyed *attr* attributes object can be reinitialized using *pthread_rwlockattr_init()*; the results of
35971 otherwise referencing the object after it has been destroyed are undefined. An implementation
35972 may cause *pthread_rwlockattr_destroy()* to set the object referenced by *attr* to an invalid value.

35973 The *pthread_rwlockattr_init()* function shall initialize a read-write lock attributes object *attr* with
35974 the default value for all of the attributes defined by the implementation.

35975 Results are undefined if *pthread_rwlockattr_init()* is called specifying an already initialized *attr*
35976 attributes object.

35977 After a read-write lock attributes object has been used to initialize one or more read-write locks,
35978 any function affecting the attributes object (including destruction) shall not affect any previously
35979 initialized read-write locks.

35980 RETURN VALUE

35981 If successful, the *pthread_rwlockattr_destroy()* and *pthread_rwlockattr_init()* functions shall return
35982 zero; otherwise, an error number shall be returned to indicate the error.

35983 ERRORS

35984 The *pthread_rwlockattr_destroy()* function may fail if:

35985 [EINVAL] The value specified by *attr* is invalid.

35986 The *pthread_rwlockattr_init()* function shall fail if:

35987 [ENOMEM] Insufficient memory exists to initialize the read-write lock attributes object.

35988 These functions shall not return an error code of [EINTR].

35989 EXAMPLES

35990 None.

35991 APPLICATION USAGE

35992 None.

35993 RATIONALE

35994 None.

35995 FUTURE DIRECTIONS

35996 None.

35997 SEE ALSO

35998 *pthread_rwlock_destroy()*, *pthread_rwlockattr_getpshared()*, *pthread_rwlockattr_setpshared()*, the
35999 Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

36000 CHANGE HISTORY

36001 First released in Issue 5.

36002 Issue 6

36003 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 36004 • The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- 36005
- 36006
- 36007 • The SEE ALSO section is updated.

36008 NAME

36009 `pthread_rwlockattr_getpshared`, `pthread_rwlockattr_setpshared` — get and set the process-
 36010 shared attribute of the read-write lock attributes object

36011 SYNOPSIS

36012 THR TSH `#include <pthread.h>`

```
36013     int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *
36014             restrict attr, int *restrict pshared);
36015     int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
36016             int pshared);
```

36017

36018 DESCRIPTION

36019 The `pthread_rwlockattr_getpshared()` function shall obtain the value of the *process-shared* attribute
 36020 from the initialized attributes object referenced by `attr`. The `pthread_rwlockattr_setpshared()`
 36021 function shall set the *process-shared* attribute in an initialized attributes object referenced by `attr`.

36022 The *process-shared* attribute shall be set to PTHREAD_PROCESS_SHARED to permit a read-
 36023 write lock to be operated upon by any thread that has access to the memory where the read-
 36024 write lock is allocated, even if the read-write lock is allocated in memory that is shared by
 36025 multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the read-
 36026 write lock shall only be operated upon by threads created within the same process as the thread
 36027 that initialized the read-write lock; if threads of differing processes attempt to operate on such a
 36028 read-write lock, the behavior is undefined. The default value of the *process-shared* attribute shall
 36029 be PTHREAD_PROCESS_PRIVATE.

36030 Additional attributes, their default values, and the names of the associated functions to get and
 36031 set those attribute values are implementation-defined.

36032 RETURN VALUE

36033 Upon successful completion, the `pthread_rwlockattr_getpshared()` function shall return zero and
 36034 store the value of the *process-shared* attribute of `attr` into the object referenced by the `pshared`
 36035 parameter. Otherwise, an error number shall be returned to indicate the error.

36036 If successful, the `pthread_rwlockattr_setpshared()` function shall return zero; otherwise, an error
 36037 number shall be returned to indicate the error.

36038 ERRORS

36039 The `pthread_rwlockattr_getpshared()` and `pthread_rwlockattr_setpshared()` functions may fail if:

36040 [EINVAL] The value specified by `attr` is invalid.

36041 The `pthread_rwlockattr_setpshared()` function may fail if:

36042 [EINVAL] The new value specified for the attribute is outside the range of legal values
 36043 for that attribute.

36044 These functions shall not return an error code of [EINTR].

36045 EXAMPLES

36046 None.

36047 APPLICATION USAGE

36048 None.

36049 RATIONALE

36050 None.

36051 FUTURE DIRECTIONS

36052 None.

36053 SEE ALSO

36054 *pthread_rwlock_destroy()*, *pthread_rwlockattr_destroy()*, *pthread_rwlockattr_init()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

36056 CHANGE HISTORY

36057 First released in Issue 5.

36058 Issue 6

36059 The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 36060 • The margin code in the SYNOPSIS is changed to THR TSH to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- 36061
- 36062
- 36063 • The DESCRIPTION notes that additional attributes are implementation-defined.
- 36064 • The SEE ALSO section is updated.

36065 The **restrict** keyword is added to the *pthread_rwlockattr_getpshared()* prototype for alignment with the ISO/IEC 9899:1999 standard.

36067 **NAME**

36068 pthread_rwlockattr_init — initialize the read-write lock attributes object

36069 **SYNOPSIS**

36070 THR #include <pthread.h>

1

36071 int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

36072

36073 **DESCRIPTION**

36074 Refer to *pthread_rwlockattr_destroy()*.

36075 NAME

36076 pthread_rwlockattr_setpshared — set the process-shared attribute of the read-write lock
36077 attributes object

36078 SYNOPSIS

36079 THR TSH #include <pthread.h>

1

```
36080     int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,  
36081             int pshared);  
36082
```

36083 DESCRIPTION

36084 Refer to *pthread_rwlockattr_getpshared()*.

36085 NAME

36086 *pthread_self* — get the calling thread ID

36087 SYNOPSIS

36088 THR *#include <pthread.h>*

36089 *pthread_t pthread_self(void);*

36090

36091 DESCRIPTION

36092 The *pthread_self()* function shall return the thread ID of the calling thread.

36093 RETURN VALUE

36094 Refer to the DESCRIPTION.

36095 ERRORS

36096 No errors are defined.

36097 The *pthread_self()* function shall not return an error code of [EINTR].

36098 EXAMPLES

36099 None.

36100 APPLICATION USAGE

36101 None.

36102 RATIONALE

36103 The *pthread_self()* function provides a capability similar to the *getpid()* function for processes
36104 and the rationale is the same: the creation call does not provide the thread ID to the created
36105 thread.

36106 FUTURE DIRECTIONS

36107 None.

36108 SEE ALSO

36109 *pthread_create()*, *pthread_equal()*, the Base Definitions volume of IEEE Std 1003.1-2001,
36110 *<pthread.h>*

36111 CHANGE HISTORY

36112 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

36113 Issue 6

36114 The *pthread_self()* function is marked as part of the Threads option.

36115 NAME

36116 **pthread_setcancelstate**, **pthread_setcanceltype**, **pthread_testcancel** — set cancelability state

36117 SYNOPSIS

36118 THR `#include <pthread.h>`

```
36119   int pthread_setcancelstate(int state, int *oldstate);  
36120   int pthread_setcanceltype(int type, int *oldtype);  
36121   void pthread_testcancel(void);  
36122
```

36123 DESCRIPTION

36124 The *pthread_setcancelstate()* function shall atomically both set the calling thread's cancelability
36125 state to the indicated *state* and return the previous cancelability state at the location referenced
36126 by *oldstate*. Legal values for *state* are PTHREAD_CANCEL_ENABLE and
36127 PTHREAD_CANCEL_DISABLE.

36128 The *pthread_setcanceltype()* function shall atomically both set the calling thread's cancelability
36129 type to the indicated *type* and return the previous cancelability type at the location referenced by
36130 *oldtype*. Legal values for *type* are PTHREAD_CANCEL_DEFERRED and
36131 PTHREAD_CANCEL_ASYNCHRONOUS.

36132 The cancelability state and type of any newly created threads, including the thread in which
36133 *main()* was first invoked, shall be PTHREAD_CANCEL_ENABLE and
36134 PTHREAD_CANCEL_DEFERRED respectively.

36135 The *pthread_testcancel()* function shall create a cancellation point in the calling thread. The
36136 *pthread_testcancel()* function shall have no effect if cancelability is disabled.

36137 RETURN VALUE

36138 If successful, the *pthread_setcancelstate()* and *pthread_setcanceltype()* functions shall return zero;
36139 otherwise, an error number shall be returned to indicate the error.

36140 ERRORS

36141 The *pthread_setcancelstate()* function may fail if:

36142 [EINVAL] The specified state is not PTHREAD_CANCEL_ENABLE or
36143 PTHREAD_CANCEL_DISABLE.

36144 The *pthread_setcanceltype()* function may fail if:

36145 [EINVAL] The specified type is not PTHREAD_CANCEL_DEFERRED or
36146 PTHREAD_CANCEL_ASYNCHRONOUS.

36147 These functions shall not return an error code of [EINTR].

36148 EXAMPLES

36149 None.

36150 APPLICATION USAGE

36151 None.

36152 RATIONALE

36153 The *pthread_setcancelstate()* and *pthread_setcanceltype()* functions control the points at which a
36154 thread may be asynchronously canceled. For cancellation control to be usable in modular
36155 fashion, some rules need to be followed.

36156 An object can be considered to be a generalization of a procedure. It is a set of procedures and
36157 global variables written as a unit and called by clients not known by the object. Objects may
36158 depend on other objects.

36159 First, cancelability should only be disabled on entry to an object, never explicitly enabled. On
36160 exit from an object, the cancelability state should always be restored to its value on entry to the
36161 object.

36162 This follows from a modularity argument: if the client of an object (or the client of an object that
36163 uses that object) has disabled cancelability, it is because the client does not want to be concerned
36164 about cleaning up if the thread is canceled while executing some sequence of actions. If an object
36165 is called in such a state and it enables cancelability and a cancellation request is pending for that
36166 thread, then the thread is canceled, contrary to the wish of the client that disabled.

36167 Second, the cancelability type may be explicitly set to either *deferred* or *asynchronous* upon entry
36168 to an object. But as with the cancelability state, on exit from an object the cancelability type
36169 should always be restored to its value on entry to the object.

36170 Finally, only functions that are cancel-safe may be called from a thread that is asynchronously
36171 cancelable.

36172 FUTURE DIRECTIONS

36173 None.

36174 SEE ALSO

36175 *pthread_cancel()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**pthread.h**>

36176 CHANGE HISTORY

36177 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

36178 Issue 6

36179 The *pthread_setcancelstate()*, *pthread_setcanceltype()*, and *pthread_testcancel()* functions are marked
36180 as part of the Threads option.

36181 **NAME**

36182 pthread_setconcurrency — set the level of concurrency

36183 **SYNOPSIS**

36184 XSI #include <pthread.h>

36185 int pthread_setconcurrency(int new_level);

36186

36187 **DESCRIPTION**

36188 Refer to *pthread_getconcurrency()*.

36189 **NAME**

36190 `pthread_setschedparam` — dynamic thread scheduling parameters access (**REALTIME**
36191 **THREADS**)

36192 **SYNOPSIS**

36193 THR TPS `#include <pthread.h>`

```
36194        int pthread_setschedparam(pthread_t thread, int policy,
36195              const struct sched_param *param);
```

36196

36197 **DESCRIPTION**

36198 Refer to *pthread_getschedparam()*.

36199 NAME

36200 pthread_setschedprio — dynamic thread scheduling parameters access (**REALTIME**
36201 **THREADS**)

36202 SYNOPSIS

```
36203 THR TPS #include <pthread.h>  
36204        int pthread_setschedprio(pthread_t thread, int prio);  
36205
```

36206 DESCRIPTION

36207 The *pthread_setschedprio()* function shall set the scheduling priority for the thread whose thread
36208 ID is given by *thread* to the value given by *prio*. See **Scheduling Policies** (on page 44) for a
36209 description on how this function call affects the ordering of the thread in the thread list for its
36210 new priority.

36211 If the *pthread_setschedprio()* function fails, the scheduling priority of the target thread shall not be
36212 changed.

36213 RETURN VALUE

36214 If successful, the *pthread_setschedprio()* function shall return zero; otherwise, an error number
36215 shall be returned to indicate the error.

36216 ERRORS

36217 The *pthread_setschedprio()* function may fail if:

36218 [EINVAL] The value of *prio* is invalid for the scheduling policy of the specified thread.
36219 [ENOTSUP] An attempt was made to set the priority to an unsupported value.
36220 [EPERM] The caller does not have the appropriate permission to set the scheduling
36221 policy of the specified thread.
36222 [EPERM] The implementation does not allow the application to modify the priority to
36223 the value specified.
36224 [ESRCH] The value specified by *thread* does not refer to an existing thread.
36225 The *pthread_setschedprio()* function shall not return an error code of [EINTR].

36226 EXAMPLES

36227 None.

36228 APPLICATION USAGE

36229 None.

36230 RATIONALE

36231 The *pthread_setschedprio()* function provides a way for an application to temporarily raise its
36232 priority and then lower it again, without having the undesired side effect of yielding to other
36233 threads of the same priority. This is necessary if the application is to implement its own
36234 strategies for bounding priority inversion, such as priority inheritance or priority ceilings. This
36235 capability is especially important if the implementation does not support the Thread Priority
36236 Protection or Thread Priority Inheritance options, but even if those options are supported it is
36237 needed if the application is to bound priority inheritance for other resources, such as
36238 semaphores.

36239 The standard developers considered that while it might be preferable conceptually to solve this
36240 problem by modifying the specification of *pthread_setschedparam()*, it was too late to make such a
36241 change, as there may be implementations that would need to be changed. Therefore, this new
36242 function was introduced.

36243 FUTURE DIRECTIONS

36244 None.

36245 SEE ALSO

36246 **Scheduling Policies** (on page 44), *pthread_getschedparam()*, the Base Definitions volume of
36247 IEEE Std 1003.1-2001, <**pthread.h**>

36248 CHANGE HISTORY

36249 First released in Issue 6. Included as a response to IEEE PASC Interpretation 1003.1 #96.

36250 **NAME**36251 `pthread_setspecific` — thread-specific data management36252 **SYNOPSIS**36253 `THR #include <pthread.h>`36254 `int pthread_setspecific(pthread_key_t key, const void *value);`

36255

36256 **DESCRIPTION**36257 Refer to *pthread_getspecific()*.

36258 NAME

36259 pthread_sigmask, sigprocmask — examine and change blocked signals

36260 SYNOPSIS

```
36261        #include <signal.h>
36262        int pthread_sigmask(int how, const sigset_t *restrict set,
36263                        sigset_t *restrict oset);
36264        CX        int sigprocmask(int how, const sigset_t *restrict set,
36265                       sigset_t *restrict oset);
36266
```

36267 DESCRIPTION

36268 THR The *pthread_sigmask()* function shall examine or change (or both) the calling thread's signal mask, regardless of the number of threads in the process. The function shall be equivalent to *sigprocmask()*, without the restriction that the call be made in a single-threaded process.

36271 In a single-threaded process, the *sigprocmask()* function shall examine or change (or both) the signal mask of the calling thread.

36273 If the argument *set* is not a null pointer, it points to a set of signals to be used to change the currently blocked set.

36275 The argument *how* indicates the way in which the set is changed, and the application shall ensure it consists of one of the following values:

36277 SIG_BLOCK The resulting set shall be the union of the current set and the signal set pointed to by *set*.

36279 SIG_SETMASK The resulting set shall be the signal set pointed to by *set*.

36280 SIG_UNBLOCK The resulting set shall be the intersection of the current set and the complement of the signal set pointed to by *set*.

36282 If the argument *oset* is not a null pointer, the previous mask shall be stored in the location pointed to by *oset*. If *set* is a null pointer, the value of the argument *how* is not significant and the thread's signal mask shall be unchanged; thus the call can be used to enquire about currently blocked signals. 2

36286 If there are any pending unblocked signals after the call to *sigprocmask()*, at least one of those signals shall be delivered before the call to *sigprocmask()* returns. 2

36288 It is not possible to block those signals which cannot be ignored. This shall be enforced by the system without causing an error to be indicated.

36290 If any of the SIGFPE, SIGILL, SIGSEGV, or SIGBUS signals are generated while they are blocked, the result is undefined, unless the signal was generated by the *kill()* function, the *sigqueue()* function, or the *raise()* function.

36293 If *sigprocmask()* fails, the thread's signal mask shall not be changed.

36294 The use of the *sigprocmask()* function is unspecified in a multi-threaded process.

36295 RETURN VALUE

36296 THR Upon successful completion *pthread_sigmask()* shall return 0; otherwise, it shall return the corresponding error number.

36298 Upon successful completion, *sigprocmask()* shall return 0; otherwise, -1 shall be returned, *errno* shall be set to indicate the error, and the process' signal mask shall be unchanged.

36300 ERRORS

- | | | |
|-------|-----|--|
| 36301 | THR | The <i>pthread_sigmask()</i> and <i>sigprocmask()</i> functions shall fail if: |
| 36302 | | [EINVAL] The value of the <i>how</i> argument is not equal to one of the defined values. |
| 36303 | THR | The <i>pthread_sigmask()</i> function shall not return an error code of [EINTR]. |

36304 EXAMPLES

36305 **Signalling in a Multi-Threaded Process**

36306 This example shows the use of *pthread_sigmask()* in order to deal with signals in a multi-
36307 threaded process. It provides a fairly general framework that could be easily adapted/extended.

```
36308 #include <stdio.h>
36309 #include <stdlib.h>
36310 #include <pthread.h>
36311 #include <signal.h>
36312 #include <string.h>
36313 #include <errno.h>
36314 ...
36315 static sigset_t signal_mask; /* signals to block */ 
36316 int main (int argc, char *argv[])
36317 {
36318     pthread_t sig_thr_id; /* signal handler thread ID */
36319     int rc; /* return code */
36320     sigemptyset (&signal_mask);
36321     sigaddset (&signal_mask, SIGINT);
36322     sigaddset (&signal_mask, SIGTERM);
36323     rc = pthread_sigmask (SIG_BLOCK, &signal_mask, NULL);
36324     if (rc != 0) {
36325         /* handle error */
36326         ...
36327     }
36328     /* any newly created threads inherit the signal mask */
36329     rc = pthread_create (&sig_thr_id, NULL, signal_thread, NULL);
36330     if (rc != 0) {
36331         /* handle error */
36332         ...
36333     }
36334     /* APPLICATION CODE */
36335     ...
36336 }
36337 void *signal_thread (void *arg)
36338 {
36339     int sig_caught; /* signal caught */
36340     int rc; /* returned code */
36341     rc = sigwait (&signal_mask, &sig_caught);
36342     if (rc != 0) {
36343         /* handle error */
36344     }
```

```

36345     switch (sig_caught)
36346     {
36347         case SIGINT: /* process SIGINT */
36348             ...
36349             break;
36350         case SIGTERM: /* process SIGTERM */
36351             ...
36352             break;
36353         default: /* should normally not happen */
36354             fprintf (stderr, "\nUnexpected signal %d\n", sig_caught);
36355             break;
36356     }
36357 }
```

36358 APPLICATION USAGE

36359 None.

36360 RATIONALE

36361 When a thread's signal mask is changed in a signal-catching function that is installed by 2
 36362 *sigaction()*, the restoration of the signal mask on return from the signal-catching function
 36363 overrides that change (see *sigaction()*). If the signal-catching function was installed with
 36364 *signal()*, it is unspecified whether this occurs.

36365 See *kill()* for a discussion of the requirement on delivery of signals.

36366 FUTURE DIRECTIONS

36367 None.

36368 SEE ALSO

36369 *sigaction()*, *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *sigpending()*,
 36370 *sigqueue()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>

36371 CHANGE HISTORY

36372 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

36373 Issue 5

36374 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

36375 The *pthread_sigmask()* function is added for alignment with the POSIX Threads Extension.

36376 Issue 6

36377 The *pthread_sigmask()* function is marked as part of the Threads option.

36378 The SYNOPSIS for *sigprocmask()* is marked as a CX extension to note that the presence of this 2
 36379 function in the <signal.h> header is an extension to the ISO C standard.

36380 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- 36381 • The DESCRIPTION is updated to explicitly state the functions which may generate the
 36382 signal.

36383 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

36384 The **restrict** keyword is added to the *pthread_sigmask()* and *sigprocmask()* prototypes for
 36385 alignment with the ISO/IEC 9899: 1999 standard.

36386 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/105 is applied, updating "process" signal 2
 36387 mask" to "thread's signal mask" in the DESCRIPTION and RATIONALE sections.

36388
36389

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/106 is applied, adding the example to the
EXAMPLES section.

2
2

36390 NAME

36391 pthread_spin_destroy, pthread_spin_init — destroy or initialize a spin lock object (**ADVANCED**
 36392 **REALTIME THREADS**)

36393 SYNOPSIS

```
36394 THR SPI #include <pthread.h>
36395     int pthread_spin_destroy(pthread_spinlock_t *lock);
36396     int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

36397

36398 DESCRIPTION

36399 The *pthread_spin_destroy()* function shall destroy the spin lock referenced by *lock* and release any
 36400 resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is
 36401 reinitialized by another call to *pthread_spin_init()*. The results are undefined if
 36402 *pthread_spin_destroy()* is called when a thread holds the lock, or if this function is called with an
 36403 uninitialized thread spin lock.

36404 The *pthread_spin_init()* function shall allocate any resources required to use the spin lock
 36405 referenced by *lock* and initialize the lock to an unlocked state.

36406 TSH If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is
 36407 PTHREAD_PROCESS_SHARED, the implementation shall permit the spin lock to be operated
 36408 upon by any thread that has access to the memory where the spin lock is allocated, even if it is
 36409 allocated in memory that is shared by multiple processes.

36410 If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is
 36411 PTHREAD_PROCESS_PRIVATE, or if the option is not supported, the spin lock shall only be
 36412 operated upon by threads created within the same process as the thread that initialized the spin
 36413 lock. If threads of differing processes attempt to operate on such a spin lock, the behavior is
 36414 undefined.

36415 The results are undefined if *pthread_spin_init()* is called specifying an already initialized spin
 36416 lock. The results are undefined if a spin lock is used without first being initialized.

36417 If the *pthread_spin_init()* function fails, the lock is not initialized and the contents of *lock* are
 36418 undefined.

36419 Only the object referenced by *lock* may be used for performing synchronization.

36420 The result of referring to copies of that object in calls to *pthread_spin_destroy()*,
 36421 *pthread_spin_lock()*, *pthread_spin_trylock()*, or *pthread_spin_unlock()* is undefined.

36422 RETURN VALUE

36423 Upon successful completion, these functions shall return zero; otherwise, an error number shall
 36424 be returned to indicate the error.

36425 ERRORS

36426 These functions may fail if:

36427 [EBUSY] The implementation has detected an attempt to initialize or destroy a spin
 36428 lock while it is in use (for example, while being used in a *pthread_spin_lock()*
 36429 call) by another thread.

36430 [EINVAL] The value specified by *lock* is invalid.

36431 The *pthread_spin_init()* function shall fail if:

36432 [EAGAIN] The system lacks the necessary resources to initialize another spin lock.

36433 [ENOMEM] Insufficient memory exists to initialize the lock.

36434 These functions shall not return an error code of [EINTR].

36435 EXAMPLES

36436 None.

36437 APPLICATION USAGE

36438 The *pthread_spin_destroy()* and *pthread_spin_init()* functions are part of the Spin Locks option
36439 and need not be provided on all implementations.

36440 RATIONALE

36441 None.

36442 FUTURE DIRECTIONS

36443 None.

36444 SEE ALSO

36445 *pthread_spin_lock()*, *pthread_spin_unlock()*, the Base Definitions volume of IEEE Std 1003.1-2001,
36446 <pthread.h>

36447 CHANGE HISTORY

36448 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

36449 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

36450 NAME

36451 pthread_spin_lock, pthread_spin_trylock — lock a spin lock object (**ADVANCED REALTIME**
36452 **THREADS**)

36453 SYNOPSIS

36454 THR SPI #include <pthread.h>

36455 int pthread_spin_lock(pthread_spinlock_t *lock);
36456 int pthread_spin_trylock(pthread_spinlock_t *lock);

36457

36458 DESCRIPTION

36459 The *pthread_spin_lock()* function shall lock the spin lock referenced by *lock*. The calling thread
36460 shall acquire the lock if it is not held by another thread. Otherwise, the thread shall spin (that is,
36461 shall not return from the *pthread_spin_lock()* call) until the lock becomes available. The results
36462 are undefined if the calling thread holds the lock at the time the call is made. The *pthread_spin_trylock()*
36463 function shall lock the spin lock referenced by *lock* if it is not held by any
36464 thread. Otherwise, the function shall fail.

36465 The results are undefined if any of these functions is called with an uninitialized spin lock.

36466 RETURN VALUE

36467 Upon successful completion, these functions shall return zero; otherwise, an error number shall
36468 be returned to indicate the error.

36469 ERRORS

36470 These functions may fail if:

36471 [EINVAL] The value specified by *lock* does not refer to an initialized spin lock object.

36472 The *pthread_spin_lock()* function may fail if:

36473 [EDEADLK] A deadlock condition was detected or the calling thread already holds the 2
36474 lock. 2

36475 The *pthread_spin_trylock()* function shall fail if:

36476 [EBUSY] A thread currently holds the lock.

36477 These functions shall not return an error code of [EINTR].

36478 EXAMPLES

36479 None.

36480 APPLICATION USAGE

36481 Applications using this function may be subject to priority inversion, as discussed in the Base
36482 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

36483 The *pthread_spin_lock()* and *pthread_spin_trylock()* functions are part of the Spin Locks option
36484 and need not be provided on all implementations.

36485 RATIONALE

36486 None.

36487 FUTURE DIRECTIONS

36488 None.

36489 SEE ALSO

36490 *pthread_spin_destroy()*, *pthread_spin_unlock()*, the Base Definitions volume of
36491 IEEE Std 1003.1-2001, <pthread.h>

36492 CHANGE HISTORY

36493 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

36494 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

36495 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/107 is applied, updating the ERRORS section so that the [EDEADLK] error includes detection of a deadlock condition. 2
36496 2

36497 NAME

36498 **pthread_spin_unlock** — unlock a spin lock object (**ADVANCED REALTIME THREADS**)

36499 SYNOPSIS

36500 THR SPI `#include <pthread.h>`

36501 `int pthread_spin_unlock(pthread_spinlock_t *lock);`

36502

36503 DESCRIPTION

36504 The *pthread_spin_unlock()* function shall release the spin lock referenced by *lock* which was
36505 locked via the *pthread_spin_lock()* or *pthread_spin_trylock()* functions. The results are undefined if
36506 the lock is not held by the calling thread. If there are threads spinning on the lock when
36507 *pthread_spin_unlock()* is called, the lock becomes available and an unspecified spinning thread
36508 shall acquire the lock.

36509 The results are undefined if this function is called with an uninitialized thread spin lock.

36510 RETURN VALUE

36511 Upon successful completion, the *pthread_spin_unlock()* function shall return zero; otherwise, an
36512 error number shall be returned to indicate the error.

36513 ERRORS

36514 The *pthread_spin_unlock()* function may fail if:

36515 [EINVAL] An invalid argument was specified.

36516 [EPERM] The calling thread does not hold the lock.

36517 This function shall not return an error code of [EINTR].

36518 EXAMPLES

36519 None.

36520 APPLICATION USAGE

36521 The *pthread_spin_unlock()* function is part of the Spin Locks option and need not be provided on
36522 all implementations.

36523 RATIONALE

36524 None.

36525 FUTURE DIRECTIONS

36526 None.

36527 SEE ALSO

36528 *pthread_spin_destroy()*, *pthread_spin_lock()*, the Base Definitions volume of IEEE Std 1003.1-2001,
36529 <**pthread.h**>

36530 CHANGE HISTORY

36531 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

36532 In the SYNOPSIS, the inclusion of <**sys/types.h**> is no longer required.

36533 **NAME**36534 `pthread_testcancel` — set cancelability state36535 **SYNOPSIS**36536 `THR #include <pthread.h>`36537 `void pthread_testcancel(void);`

36538

36539 **DESCRIPTION**36540 Refer to *pthread_setcancelstate()*.

36541 NAME

36542 ptsname — get name of the slave pseudo-terminal device

36543 SYNOPSIS

36544 XSI `#include <stdlib.h>`

36545 `char *ptsname(int fildes);`

36546

36547 DESCRIPTION

36548 The *ptsname()* function shall return the name of the slave pseudo-terminal device associated
36549 with a master pseudo-terminal device. The *fildes* argument is a file descriptor that refers to the
36550 master device. The *ptsname()* function shall return a pointer to a string containing the pathname
36551 of the corresponding slave device.

36552 The *ptsname()* function need not be reentrant. A function that is not required to be reentrant is
36553 not required to be thread-safe.

36554 RETURN VALUE

36555 Upon successful completion, *ptsname()* shall return a pointer to a string which is the name of the
36556 pseudo-terminal slave device. Upon failure, *ptsname()* shall return a null pointer. This could
36557 occur if *fildes* is an invalid file descriptor or if the slave device name does not exist in the file
36558 system.

36559 ERRORS

36560 No errors are defined.

36561 EXAMPLES

36562 None.

36563 APPLICATION USAGE

36564 The value returned may point to a static data area that is overwritten by each call to *ptsname()*.

36565 RATIONALE

36566 None.

36567 FUTURE DIRECTIONS

36568 None.

36569 SEE ALSO

36570 `grantpt()`, `open()`, `ttynname()`, `unlockpt()`, the Base Definitions volume of IEEE Std 1003.1-2001,
36571 `<stdlib.h>`

36572 CHANGE HISTORY

36573 First released in Issue 4, Version 2.

36574 Issue 5

36575 Moved from X/OPEN UNIX extension to BASE.

36576 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

36577 NAME

36578 **putc** — put a byte on a stream

36579 SYNOPSIS

```
36580        #include <stdio.h>
36581        int putc(int c, FILE *stream);
```

36582 DESCRIPTION

36583 CX The functionality described on this reference page is aligned with the ISO C standard. Any
36584 conflict between the requirements described here and the ISO C standard is unintentional. This
36585 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

36586 The *putc()* function shall be equivalent to *fputc()*, except that if it is implemented as a macro it
36587 may evaluate *stream* more than once, so the argument should never be an expression with side
36588 effects.

36589 RETURN VALUE

36590 Refer to *fputc()*.

36591 ERRORS

36592 Refer to *fputc()*.

36593 EXAMPLES

36594 None.

36595 APPLICATION USAGE

36596 Since it may be implemented as a macro, *putc()* may treat a *stream* argument with side effects
36597 incorrectly. In particular, *putc(c,*f++)* does not necessarily work correctly. Therefore, use of this
36598 function is not recommended in such situations; *fputc()* should be used instead.

36599 RATIONALE

36600 None.

36601 FUTURE DIRECTIONS

36602 None.

36603 SEE ALSO

36604 *fputc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

36605 CHANGE HISTORY

36606 First released in Issue 1. Derived from Issue 1 of the SVID.

36607 NAME

36608 `putc_unlocked` — stdio with explicit client locking

36609 SYNOPSIS

36610 TSF `#include <stdio.h>`

36611 `int putc_unlocked(int c, FILE *stream);`

36612

36613 DESCRIPTION

36614 Refer to *getc_unlocked()*.

36615 NAME

36616 `putchar` — put a byte on a stdout stream

36617 SYNOPSIS

```
36618        #include <stdio.h>
36619        int putchar(int c);
```

36620 DESCRIPTION

36621 CX 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.

36624 The function call `putchar(c)` shall be equivalent to `putc(c,stdout)`.

36625 RETURN VALUE

36626 Refer to `fputc()`.

36627 ERRORS

36628 Refer to `fputc()`.

36629 EXAMPLES

36630 None.

36631 APPLICATION USAGE

36632 None.

36633 RATIONALE

36634 None.

36635 FUTURE DIRECTIONS

36636 None.

36637 SEE ALSO

36638 `putc()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`

36639 CHANGE HISTORY

36640 First released in Issue 1. Derived from Issue 1 of the SVID.

36641 NAME

36642 **putchar_unlocked** — stdio with explicit client locking

36643 SYNOPSIS

36644 TSF

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

36645

```
int putchar_unlocked(int c);
```

36646

36647 DESCRIPTION

36648 Refer to *getc_unlocked()*.

36649 NAME

36650 putenv — change or add a value to an environment

36651 SYNOPSIS

36652 XSI #include <stdlib.h>

36653 int putenv(char *string);

36654

36655 DESCRIPTION

36656 The *putenv()* function shall use the *string* argument to set environment variable values. The
36657 *string* argument should point to a string of the form "*name=value*". The *putenv()* function shall
36658 make the value of the environment variable *name* equal to *value* by altering an existing variable
36659 or creating a new one. In either case, the string pointed to by *string* shall become part of the
36660 environment, so altering the string shall change the environment. The space used by *string* is no
36661 longer used once a new string which defines *name* is passed to *putenv()*.
1
1
1

36662 The *putenv()* function need not be reentrant. A function that is not required to be reentrant is not
36663 required to be thread-safe.

36664 RETURN VALUE

36665 Upon successful completion, *putenv()* shall return 0; otherwise, it shall return a non-zero value
36666 and set *errno* to indicate the error.

36667 ERRORS

36668 The *putenv()* function may fail if:

36669 [ENOMEM] Insufficient memory was available.

36670 EXAMPLES

36671 **Changing the Value of an Environment Variable**

36672 The following example changes the value of the *HOME* environment variable to the value
36673 **/usr/home**.

```
36674 #include <stdlib.h>
36675 ...
36676 static char *var = "HOME=/usr/home";
36677 int ret;
36678 ret = putenv(var);
```

36679 APPLICATION USAGE

36680 The *putenv()* function manipulates the environment pointed to by *environ*, and can be used in
36681 conjunction with *getenv()*.

36682 See *exec*, for restrictions on changing the environment in multi-threaded applications.
1

36683 This routine may use *malloc()* to enlarge the environment.

36684 A potential error is to call *putenv()* with an automatic variable as the argument, then return from
36685 the calling function while *string* is still part of the environment.

36686 The *setenv()* function is preferred over this function.

36687 RATIONALE

36688 The standard developers noted that *putenv()* is the only function available to add to the
36689 environment without permitting memory leaks.

36690 FUTURE DIRECTIONS

36691 None.

36692 SEE ALSO

36693 *exec, getenv(), malloc(), setenv()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

36694 CHANGE HISTORY

36695 First released in Issue 1. Derived from Issue 1 of the SVID.

36696 Issue 5

36697 The type of the argument to this function is changed from **const char *** to **char ***. This was
36698 indicated as a FUTURE DIRECTION in previous issues.

36699 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

36700 Issue 6

36701 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/48 is applied, clarifying wording in the 1
36702 DESCRIPTION and adding a new paragraph into APPLICATION USAGE referring readers to 1
36703 exec. 1

36704 NAME

36705 putmsg, putpmsg — send a message on a STREAM (STREAMS)

36706 SYNOPSIS

36707 XSR #include <stropts.h>

```
36708 int putmsg(int fildes, const struct strbuf *ctlptr,
36709     const struct strbuf *dataptr, int flags);
36710 int putpmsg(int fildes, const struct strbuf *ctlptr,
36711     const struct strbuf *dataptr, int band, int flags);
```

36712

36713 DESCRIPTION

36714 The *putmsg()* function shall create a message from a process buffer(s) and send the message to a
 36715 STREAMS file. The message may contain either a data part, a control part, or both. The data and
 36716 control parts are distinguished by placement in separate buffers, as described below. The
 36717 semantics of each part are defined by the STREAMS module that receives the message.

36718 The *putpmsg()* function is equivalent to *putmsg()*, except that the process can send messages in
 36719 different priority bands. Except where noted, all requirements on *putmsg()* also pertain to
 36720 *putpmsg()*.

36721 The *fildes* argument specifies a file descriptor referencing an open STREAM. The *ctlptr* and
 36722 *dataptr* arguments each point to a **strbuf** structure.

36723 The *ctlptr* argument points to the structure describing the control part, if any, to be included in
 36724 the message. The *buf* member in the **strbuf** structure points to the buffer where the control
 36725 information resides, and the *len* member indicates the number of bytes to be sent. The *maxlen*
 36726 member is not used by *putmsg()*. In a similar manner, the argument *dataptr* specifies the data, if
 36727 any, to be included in the message. The *flags* argument indicates what type of message should be
 36728 sent and is described further below.

36729 To send the data part of a message, the application shall ensure that *dataptr* is not a null pointer
 36730 and the *len* member of *dataptr* is 0 or greater. To send the control part of a message, the
 36731 application shall ensure that the corresponding values are set for *ctlptr*. No data (control) part
 36732 shall be sent if either *dataptr*(*ctlptr*) is a null pointer or the *len* member of *dataptr*(*ctlptr*) is set to
 36733 -1.

36734 For *putmsg()*, if a control part is specified and *flags* is set to RS_HIPRI, a high priority message
 36735 shall be sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg()* shall fail and
 36736 set *errno* to [EINVAL]. If *flags* is set to 0, a normal message (priority band equal to 0) shall be
 36737 sent. If a control part and data part are not specified and *flags* is set to 0, no message shall be
 36738 sent and 0 shall be returned.

36739 For *putpmsg()*, the flags are different. The *flags* argument is a bitmask with the following
 36740 mutually-exclusive flags defined: MSG_HIPRI and MSG_BAND. If *flags* is set to 0, *putpmsg()*
 36741 shall fail and set *errno* to [EINVAL]. If a control part is specified and *flags* is set to MSG_HIPRI
 36742 and *band* is set to 0, a high-priority message shall be sent. If *flags* is set to MSG_HIPRI and either
 36743 no control part is specified or *band* is set to a non-zero value, *putpmsg()* shall fail and set *errno* to
 36744 [EINVAL]. If *flags* is set to MSG_BAND, then a message shall be sent in the priority band
 36745 specified by *band*. If a control part and data part are not specified and *flags* is set to MSG_BAND,
 36746 no message shall be sent and 0 shall be returned.

36747 The *putmsg()* function shall block if the STREAM write queue is full due to internal flow control
 36748 conditions, with the following exceptions:

- 36749 • For high-priority messages, *putmsg()* shall not block on this condition and continues
 36750 processing the message.

- 36751 • For other messages, *putmsg()* shall not block but shall fail when the write queue is full and
36752 O_NONBLOCK is set.

36753 The *putmsg()* function shall also block, unless prevented by lack of internal resources, while
36754 waiting for the availability of message blocks in the STREAM, regardless of priority or whether
36755 O_NONBLOCK has been specified. No partial message shall be sent.

36756 RETURN VALUE

36757 Upon successful completion, *putmsg()* and *putpmsg()* shall return 0; otherwise, they shall return
36758 -1 and set *errno* to indicate the error.

36759 ERRORS

36760 The *putmsg()* and *putpmsg()* functions shall fail if:

- 36761 [EAGAIN] A non-priority message was specified, the O_NONBLOCK flag is set, and the
36762 STREAM write queue is full due to internal flow control conditions; or buffers
36763 could not be allocated for the message that was to be created.
- 36764 [EBADF] *fildes* is not a valid file descriptor open for writing.
- 36765 [EINTR] A signal was caught during *putmsg()*.
- 36766 [EINVAL] An undefined value is specified in *flags*, or *flags* is set to RS_HIPRI or
36767 MSG_HIPRI and no control part is supplied, or the STREAM or multiplexer
36768 referenced by *fildes* is linked (directly or indirectly) downstream from a
36769 multiplexer, or *flags* is set to MSG_HIPRI and *band* is non-zero (for *putpmsg()*
36770 only).
- 36771 [ENOSR] Buffers could not be allocated for the message that was to be created due to
36772 insufficient STREAMS memory resources.
- 36773 [ENOSTR] A STREAM is not associated with *fildes*.
- 36774 [ENXIO] A hangup condition was generated downstream for the specified STREAM.
- 36775 [EPIPE] or [EIO] The *fildes* argument refers to a STREAMS-based pipe and the other end of the
36776 pipe is closed. A SIGPIPE signal is generated for the calling thread.
- 36777 [ERANGE] The size of the data part of the message does not fall within the range
36778 specified by the maximum and minimum packet sizes of the topmost
36779 STREAM module. This value is also returned if the control part of the message
36780 is larger than the maximum configured size of the control part of a message,
36781 or if the data part of a message is larger than the maximum configured size of
36782 the data part of a message.

36783 In addition, *putmsg()* and *putpmsg()* shall fail if the STREAM head had processed an
36784 asynchronous error before the call. In this case, the value of *errno* does not reflect the result of
36785 *putmsg()* or *putpmsg()*, but reflects the prior error.

36786 EXAMPLES

36787 **Sending a High-Priority Message**

36788 The value of *fd* is assumed to refer to an open STREAMS file. This call to *putmsg()* does the
36789 following:

- 36790 1. Creates a high-priority message with a control part and a data part, using the buffers
36791 pointed to by *ctrlbuf* and *databuf*, respectively.
- 36792 2. Sends the message to the STREAMS file identified by *fd*.

```
36793 #include <stropts.h>
36794 #include <string.h>
36795 ...
36796 int fd;
36797 char *ctrlbuf = "This is the control part";
36798 char *databuf = "This is the data part";
36799 struct strbuf ctrl;
36800 struct strbuf data;
36801 int ret;
36802 ctrl.buf = ctrlbuf;
36803 ctrl.len = strlen(ctrlbuf);
36804 data.buf = databuf;
36805 data.len = strlen(databuf);
36806 ret = putmsg(fd, &ctrl, &data, MSG_HIPRI);
```

36807 **Using putpmsg()**

36808 This example has the same effect as the previous example. In this example, however, the
36809 *putpmsg()* function creates and sends the message to the STREAMS file.

```
36810 #include <stropts.h>
36811 #include <string.h>
36812 ...
36813 int fd;
36814 char *ctrlbuf = "This is the control part";
36815 char *databuf = "This is the data part";
36816 struct strbuf ctrl;
36817 struct strbuf data;
36818 int ret;
36819 ctrl.buf = ctrlbuf;
36820 ctrl.len = strlen(ctrlbuf);
36821 data.buf = databuf;
36822 data.len = strlen(databuf);
36823 ret = putpmsg(fd, &ctrl, &data, 0, MSG_HIPRI);
```

36824 APPLICATION USAGE

36825 None.

36826 RATIONALE

36827 None.

36828 FUTURE DIRECTIONS

36829 None.

36830 SEE ALSO

36831 Section 2.6 (on page 38), *getmsg()*, *poll()*, *read()*, *write()*, the Base Definitions volume of
36832 IEEE Std 1003.1-2001, <stropts.h>

36833 CHANGE HISTORY

36834 First released in Issue 4, Version 2.

36835 Issue 5

36836 Moved from X/OPEN UNIX extension to BASE.

36837 The following text is removed from the DESCRIPTION: “The STREAM head guarantees that the
36838 control part of a message generated by *putmsg()* is at least 64 bytes in length”.

36839 Issue 6

36840 This function is marked as part of the XSI STREAMS Option Group.

36841 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

36842 NAME

36843 puts — put a string on standard output

36844 SYNOPSIS

36845 #include <stdio.h>

36846 int puts(const char *s);

36847 DESCRIPTION

36848 CX 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.

36851 The *puts()* function shall write the string pointed to by *s*, followed by a <newline>, to the standard output stream *stdout*. The terminating null byte shall not be written.

36853 CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful execution of *puts()* and the next successful completion of a call to *fflush()* or *fclose()* on the same stream or a call to *exit()* or *abort()*.

36856 RETURN VALUE

36857 Upon successful completion, *puts()* shall return a non-negative number. Otherwise, it shall
36858 CX return EOF, shall set an error indicator for the stream, and *errno* shall be set to indicate the error.

36859 ERRORS

36860 Refer to *fputc()*.

36861 EXAMPLES**36862 Printing to Standard Output**

36863 The following example gets the current time, converts it to a string using *localtime()* and
36864 *asctime()*, and prints it to standard output using *puts()*. It then prints the number of minutes to
36865 an event for which it is waiting.

```
36866       #include <time.h>
36867       #include <stdio.h>
36868       ...
36869       time_t now;
36870       int minutes_to_event;
36871       ...
36872       time(&now);
36873       printf("The time is ");
36874       puts(asctime(localtime(&now)));
36875       printf("There are %d minutes to the event.\n",
36876            minutes_to_event);
36877       ...
```

36878 APPLICATION USAGE

36879 The *puts()* function appends a <newline>, while *fputs()* does not.

36880 RATIONALE

36881 None.

36882 FUTURE DIRECTIONS

36883 None.

36884 SEE ALSO

36885 *fopen()*, *fputs()*, *putc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

36886 CHANGE HISTORY

36887 First released in Issue 1. Derived from Issue 1 of the SVID.

36888 Issue 6

36889 Extensions beyond the ISO C standard are marked.

36890 **NAME**36891 **pututxline** — put an entry into the user accounting database36892 **SYNOPSIS**36893 XSI

```
#include <utmpx.h>
```

36894

```
struct utmpx *pututxline(const struct utmpx *utmpx);
```

36895

36896 **DESCRIPTION**36897 Refer to *endutxent()*.

36898 NAME

36899 putwc — put a wide character on a stream

36900 SYNOPSIS

```
36901 #include <stdio.h>
36902 #include <wchar.h>
36903 wint_t putwc(wchar_t wc, FILE *stream);
```

36904 DESCRIPTION

36905 CX The functionality described on this reference page is aligned with the ISO C standard. Any
36906 conflict between the requirements described here and the ISO C standard is unintentional. This
36907 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

36908 The *putwc()* function shall be equivalent to *fputwc()*, except that if it is implemented as a macro
36909 it may evaluate *stream* more than once, so the argument should never be an expression with side
36910 effects.

36911 RETURN VALUE

36912 Refer to *fputwc()*.

36913 ERRORS

36914 Refer to *fputwc()*.

36915 EXAMPLES

36916 None.

36917 APPLICATION USAGE

36918 Since it may be implemented as a macro, *putwc()* may treat a *stream* argument with side effects
36919 incorrectly. In particular, *putwc(wc,*f++)* need not work correctly. Therefore, use of this function
36920 is not recommended; *fputwc()* should be used instead.

36921 RATIONALE

36922 None.

36923 FUTURE DIRECTIONS

36924 None.

36925 SEE ALSO

36926 *fputwc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>, <wchar.h>

36927 CHANGE HISTORY

36928 First released as a World-wide Portability Interface in Issue 4.

36929 Issue 5

36930 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc*
36931 is changed from *wint_t* to *wchar_t*.

36932 The Optional Header (OH) marking is removed from <stdio.h>.

36933 NAME

36934 putwchar — put a wide character on a stdout stream

36935 SYNOPSIS

36936 #include <wchar.h>

36937 wint_t putwchar(wchar_t wc);

36938 DESCRIPTION

36939 CX 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.

36942 The function call *putwchar(wc)* shall be equivalent to *putwc(wc,stdout)*.

36943 RETURN VALUE

36944 Refer to *fputwc()*.

36945 ERRORS

36946 Refer to *fputwc()*.

36947 EXAMPLES

36948 None.

36949 APPLICATION USAGE

36950 None.

36951 RATIONALE

36952 None.

36953 FUTURE DIRECTIONS

36954 None.

36955 SEE ALSO

36956 *fputwc()*, *putwc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

36957 CHANGE HISTORY

36958 First released in Issue 4.

36959 Issue 5

36960 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc* is changed from *wint_t* to *wchar_t*.

36962 NAME

36963 pwrite — write on a file

36964 SYNOPSIS

36965 #include <unistd.h>

36966 XSI `ssize_t pwrite(int fildes, const void *buf, size_t nbyte,`
36967 `off_t offset);`

36968

36969 DESCRIPTION

36970 Refer to *write()*.

36971 NAME

36972 **qsort** — sort a table of data

36973 SYNOPSIS

```
36974     #include <stdlib.h>
36975     void qsort(void *base, size_t nel, size_t width,
36976                  int (*compar)(const void *, const void *));
```

36977 DESCRIPTION

36978 cx 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.
36979
36980

The *qsort()* function shall sort an array of *nel* objects, the initial element of which is pointed to by *base*. The size of each object, in bytes, is specified by the *width* argument. If the *nel* argument has the value zero, the comparison function pointed to by *compar* shall not be called and no rearrangement shall take place.

36985 The application shall ensure that the comparison function pointed to by *compar* does not alter the
36986 contents of the array. The implementation may reorder elements of the array between calls to the
36987 comparison function, but shall not alter the contents of any individual element.

When the same objects (consisting of width bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, they shall define a total ordering on the array.

The contents of the array shall be sorted in ascending order according to a comparison function. The *compar* argument is a pointer to the comparison function, which is called with two arguments that point to the elements being compared. The application shall ensure that the function returns an integer less than, equal to, or greater than 0, if the first argument is considered respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is unspecified.

36997 RETURN VALUE

36998 The *qsort()* function shall not return a value.

36999 ERRORS

37000 No errors are defined.

37001 EXAMPLES

37002 None.

37003 APPLICATION USAGE

37004 The comparison function need not compare every byte, so arbitrary data may be contained in
37005 the elements in addition to the values being compared.

37006 RATIONALE

The requirement that each argument (hereafter referred to as p) to the comparison function is a pointer to elements of the array implies that for every call, for each argument separately, all of the following expressions are nonzero:

```
37010      ((char *)p - (char *)base) % width == 0  
37011      (char *)p >= (char *)base  
37012      (char *)p < (char *)base + nel * width
```

37013 FUTURE DIRECTIONS

37014 None.

37015 SEE ALSO

37016 The Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

37017 CHANGE HISTORY

37018 First released in Issue 1. Derived from Issue 1 of the SVID.

37019 Issue 6

37020 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

37021 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/49 is applied, adding the last sentence to 1

37022 the first non-shaded paragraph in the DESCRIPTION, and the following two paragraphs. The 1

37023 RATIONALE is also updated. These changes are for alignment with the ISO C standard. 1

37024 NAME

37025 raise — send a signal to the executing process

37026 SYNOPSIS

37027 #include <signal.h>
37028 int raise(int sig);

37029 DESCRIPTION

37030 CX 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.

37033 CX The *raise()* function shall send the signal *sig* to the executing thread or process. If a signal handler is called, the *raise()* function shall not return until after the signal handler does.

37035 THR If the implementation supports the Threads option, the effect of the *raise()* function shall be equivalent to calling:

37037 `pthread_kill(pthread_self(), sig);`

37038

37039 CX Otherwise, the effect of the *raise()* function shall be equivalent to calling:

37040 `kill(getpid(), sig);`

37041

37042 RETURN VALUE

37043 CX Upon successful completion, 0 shall be returned. Otherwise, a non-zero value shall be returned and *errno* shall be set to indicate the error.

37045 ERRORS

37046 The *raise()* function shall fail if:

37047 CX [EINVAL] The value of the *sig* argument is an invalid signal number.

37048 EXAMPLES

37049 None.

37050 APPLICATION USAGE

37051 None.

37052 RATIONALE

37053 The term “thread” is an extension to the ISO C standard.

37054 FUTURE DIRECTIONS

37055 None.

37056 SEE ALSO

37057 `kill()`, `sigaction()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`,
37058 `<sys/types.h>`

37059 CHANGE HISTORY

37060 First released in Issue 4. Derived from the ANSI C standard.

37061 Issue 5

37062 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

37063 **Issue 6**

37064 Extensions beyond the ISO C standard are marked.

37065 The following new requirements on POSIX implementations derive from alignment with the
37066 Single UNIX Specification:

- 37067 • In the RETURN VALUE section, the requirement to set *errno* on error is added.
- 37068 • The [EINVAL] error condition is added.

37069 **NAME**

37070 rand, rand_r, srand — pseudo-random number generator

37071 **SYNOPSIS**

37072 #include <stdlib.h>

37073 int rand(void);

37074 TSF int rand_r(unsigned *seed);

37075 void srand(unsigned seed);

37076 **DESCRIPTION**

37077 CX 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.

37080 The *rand()* function shall compute a sequence of pseudo-random integers in the range [0,{RAND_MAX}] with a period of at least 2^{32} .

37082 CX The *rand()* function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

37084 TSF The *rand_r()* function shall compute a sequence of pseudo-random integers in the range [0,{RAND_MAX}]. (The value of the {RAND_MAX} macro shall be at least 32 767.)

37086 If *rand_r()* is called with the same initial value for the object pointed to by *seed* and that object is not modified between successive returns and calls to *rand_r()*, the same sequence shall be generated.

37089 The *srand()* function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to *rand()*. If *srand()* is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If *rand()* is called before any calls to *srand()* are made, the same sequence shall be generated as when *srand()* is first called with a seed value of 1.

37094 The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-2001 calls *rand()* or *srand()*.

37096 **RETURN VALUE**

37097 The *rand()* function shall return the next pseudo-random number in the sequence.

37098 TSF The *rand_r()* function shall return a pseudo-random integer.

37099 The *srand()* function shall not return a value.

37100 **ERRORS**

37101 No errors are defined.

37102 **EXAMPLES**37103 **Generating a Pseudo-Random Number Sequence**

37104 The following example demonstrates how to generate a sequence of pseudo-random numbers.

```
37105 #include <stdio.h>
37106 #include <stdlib.h>
37107 ...
37108     long count, i;
37109     char *keystr;
37110     int elementlen, len;
37111     char c;
```

```
37112     ...
37113     /* Initial random number generator. */
37114     srand(1);
37115
37116     /* Create keys using only lowercase characters */
37117     len = 0;
37118     for (i=0; i<count; i++) {
37119         while (len < elementlen) {
37120             c = (char) (rand() % 128);
37121             if (islower(c))
37122                 keystr[len++] = c;
37123         }
37124         keystr[len] = '\0';
37125         printf("%s Element%0*ld\n", keystr, elementlen, i);
37126         len = 0;
37127     }
```

37127 Generating the Same Sequence on Different Machines

37128 The following code defines a pair of functions that could be incorporated into applications
37129 wishing to ensure that the same sequence of numbers is generated across different machines.

```
37130 static unsigned long next = 1;
37131 int myrand(void) /* RAND_MAX assumed to be 32767. */
37132 {
37133     next = next * 1103515245 + 12345;
37134     return((unsigned)(next/65536) % 32768);
37135 }
37136 void mysrand(unsigned seed)
37137 {
37138     next = seed;
37139 }
```

37140 APPLICATION USAGE

37141 The *drand48()* function provides a much more elaborate random number generator.

37142 The limitations on the amount of state that can be carried between one function call and another
37143 mean the *rand_r()* function can never be implemented in a way which satisfies all of the
37144 requirements on a pseudo-random number generator. Therefore this function should be avoided
37145 whenever non-trivial requirements (including safety) have to be fulfilled.

37146 RATIONALE

37147 The ISO C standard *rand()* and *srand()* functions allow per-process pseudo-random streams
37148 shared by all threads. Those two functions need not change, but there has to be mutual-
37149 exclusion that prevents interference between two threads concurrently accessing the random
37150 number generator.

37151 With regard to *rand()*, there are two different behaviors that may be wanted in a multi-threaded
37152 program:

- 37153 1. A single per-process sequence of pseudo-random numbers that is shared by all threads
37154 that call *rand()*
- 37155 2. A different sequence of pseudo-random numbers for each thread that calls *rand()*

37156 This is provided by the modified thread-safe function based on whether the seed value is global
37157 to the entire process or local to each thread.

37158 This does not address the known deficiencies of the *rand()* function implementations, which
37159 have been approached by maintaining more state. In effect, this specifies new thread-safe forms
37160 of a deficient function.

37161 FUTURE DIRECTIONS

37162 None.

37163 SEE ALSO

37164 *drand48()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

37165 CHANGE HISTORY

37166 First released in Issue 1. Derived from Issue 1 of the SVID.

37167 Issue 5

37168 The *rand_r()* function is included for alignment with the POSIX Threads Extension.

37169 A note indicating that the *rand()* function need not be reentrant is added to the DESCRIPTION.

37170 Issue 6

37171 Extensions beyond the ISO C standard are marked.

37172 The *rand_r()* function is marked as part of the Thread-Safe Functions option.

37173 NAME

37174 random — generate pseudo-random number

37175 SYNOPSIS

37176 XSI #include <stdlib.h>

37177 long random(void);

37178

37179 DESCRIPTION

37180 Refer to *initstate()*.

37181 NAME

37182 pread, read — read from a file

37183 SYNOPSIS

37184 #include <unistd.h>

37185 XSI ssize_t pread(int *fildes*, void **buf*, size_t *nbyte*, off_t *offset*);
37186 ssize_t read(int *fildes*, void **buf*, size_t *nbyte*);

37187 DESCRIPTION

37188 The *read()* function shall attempt to read *nbyte* bytes from the file associated with the open file descriptor, *fildes*, into the buffer pointed to by *buf*. The behavior of multiple concurrent reads on the same pipe, FIFO, or terminal device is unspecified.

37191 Before any action described below is taken, and if *nbyte* is zero, the *read()* function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the *read()* function shall return zero and have no other results.

37194 On files that support seeking (for example, a regular file), the *read()* shall start at a position in the file given by the file offset associated with *fildes*. The file offset shall be incremented by the number of bytes actually read.

37197 Files that do not support seeking—for example, terminals—always read from the current position. The value of a file offset associated with such a file is undefined.

37199 No data transfer shall occur past the current end-of-file. If the starting position is at or after the end-of-file, 0 shall be returned. If the file refers to a device special file, the result of subsequent *read()* requests is implementation-defined.

37202 If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-defined.

37203 When attempting to read from an empty pipe or FIFO:

- If no process has the pipe open for writing, *read()* shall return 0 to indicate end-of-file.
- If some process has the pipe open for writing and O_NONBLOCK is set, *read()* shall return -1 and set *errno* to [EAGAIN].
- If some process has the pipe open for writing and O_NONBLOCK is clear, *read()* shall block the calling thread until some data is written or the pipe is closed by all processes that had the pipe open for writing.

37210 When attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and has no data currently available:

- If O_NONBLOCK is set, *read()* shall return -1 and set *errno* to [EAGAIN].
- If O_NONBLOCK is clear, *read()* shall block the calling thread until some data becomes available.
- The use of the O_NONBLOCK flag has no effect if there is some data available.

37216 The *read()* function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, *read()* shall return bytes with value 0. For example, *Iseek()* allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data shall return bytes with value 0 until data is written into the gap.

37221 Upon successful completion, where *nbyte* is greater than 0, *read()* shall mark for update the *st_atime* field of the file, and shall return the number of bytes read. This number shall never be greater than *nbyte*. The value returned may be less than *nbyte* if the number of bytes left in the

37224 file is less than *nbyte*, if the *read()* request was interrupted by a signal, or if the file is a pipe or
37225 FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For
37226 example, a *read()* from a file associated with a terminal may return one typed line of data.

37227 If a *read()* is interrupted by a signal before it reads any data, it shall return -1 with *errno* set to
37228 [EINTR].

37229 If a *read()* is interrupted by a signal after it has successfully read some data, it shall return the
37230 number of bytes read.

37231 For regular files, no data transfer shall occur past the offset maximum established in the open
37232 file description associated with *fd*.

37233 If *fd* refers to a socket, *read()* shall be equivalent to *recv()* with no flags set.

37234 SIO If the O_DSYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor
37235 shall complete as defined by synchronized I/O data integrity completion. If the O_SYNC and
37236 O_RSYNC bits have been set, read I/O operations on the file descriptor shall complete as
37237 defined by synchronized I/O file integrity completion.

37238 SHM If *fd* refers to a shared memory object, the result of the *read()* function is unspecified.

37239 TYM If *fd* refers to a typed memory object, the result of the *read()* function is unspecified.

37240 XSR A *read()* from a STREAMS file can read data in three different modes: *byte-stream* mode,
37241 *message-nondiscard* mode, and *message-discard* mode. The default shall be *byte-stream* mode. This
37242 can be changed using the I_SRDOPT *ioctl()* request, and can be tested with I_GRDOPT *ioctl()*.
37243 In *byte-stream* mode, *read()* shall retrieve data from the STREAM until as many bytes as were
37244 requested are transferred, or until there is no more data to be retrieved. *Byte-stream* mode
37245 ignores message boundaries.

37246 In STREAMS *message-nondiscard* mode, *read()* shall retrieve data until as many bytes as were
37247 requested are transferred, or until a message boundary is reached. If *read()* does not retrieve all
37248 the data in a message, the remaining data shall be left on the STREAM, and can be retrieved by
37249 the next *read()* call. *Message-discard* mode also retrieves data until as many bytes as were
37250 requested are transferred, or a message boundary is reached. However, unread data remaining
37251 in a message after the *read()* returns shall be discarded, and shall not be available for a
37252 subsequent *read()*, *getmsg()*, or *getpmsg()* call.

37253 How *read()* handles zero-byte STREAMS messages is determined by the current read mode
37254 setting. In *byte-stream* mode, *read()* shall accept data until it has read *nbyte* bytes, or until there
37255 is no more data to read, or until a zero-byte message block is encountered. The *read()* function
37256 shall then return the number of bytes read, and place the zero-byte message back on the
37257 STREAM to be retrieved by the next *read()*, *getmsg()*, or *getpmsg()*. In *message-nondiscard* mode
37258 or *message-discard* mode, a zero-byte message shall return 0 and the message shall be removed
37259 from the STREAM. When a zero-byte message is read as the first message on a STREAM, the
37260 message shall be removed from the STREAM and 0 shall be returned, regardless of the read
37261 mode.

37262 A *read()* from a STREAMS file shall return the data in the message at the front of the STREAM
37263 head read queue, regardless of the priority band of the message.

37264 By default, STREAMS are in control-normal mode, in which a *read()* from a STREAMS file can
37265 only process messages that contain a data part but do not contain a control part. The *read()* shall
37266 fail if a message containing a control part is encountered at the STREAM head. This default
37267 action can be changed by placing the STREAM in either control-data mode or control-discard
37268 mode with the I_SRDOPT *ioctl()* command. In control-data mode, *read()* shall convert any
37269 control part to data and pass it to the application before passing any data part originally present

37270 in the same message. In control-discard mode, *read()* shall discard message control parts but
37271 return to the process any data part in the message.

37272 In addition, *read()* shall fail if the STREAM head had processed an asynchronous error before the
37273 call. In this case, the value of *errno* shall not reflect the result of *read()*, but reflect the prior error.
37274 If a hangup occurs on the STREAM being read, *read()* shall continue to operate normally until
37275 the STREAM head read queue is empty. Thereafter, it shall return 0.

37276 XSI The *pread()* function shall be equivalent to *read()*, except that it shall read from a given position
37277 in the file without changing the file pointer. The first three arguments to *pread()* are the same as
37278 *read()* with the addition of a fourth argument *offset* for the desired position inside the file. An
37279 attempt to perform a *pread()* on a file that is incapable of seeking shall result in an error.

37280 RETURN VALUE

37281 XSI Upon successful completion, *read()* and *pread()* shall return a non-negative integer indicating the
37282 number of bytes actually read. Otherwise, the functions shall return -1 and set *errno* to indicate
37283 the error.

37284 ERRORS

37285 XSI The *read()* and *pread()* functions shall fail if:

37286 [EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the thread would be 2
37287 delayed.

37288 [EBADF] The *fd* argument is not a valid file descriptor open for reading.

37289 XSR [EBADMSG] The file is a STREAM file that is set to control-normal mode and the message
37290 waiting to be read includes a control part.

37291 [EINTR] The read operation was terminated due to the receipt of a signal, and no data
37292 was transferred.

37293 XSR [EINVAL] The STREAM or multiplexer referenced by *fd* is linked (directly or
37294 indirectly) downstream from a multiplexer.

37295 [EIO] The process is a member of a background process attempting to read from its
37296 controlling terminal, the process is ignoring or blocking the SIGTTIN signal,
37297 or the process group is orphaned. This error may also be generated for
37298 implementation-defined reasons.

37299 XSI [EISDIR] The *fd* argument refers to a directory and the implementation does not
37300 allow the directory to be read using *read()* or *pread()*. The *readdir()* function
37301 should be used instead.

37302 [EOVERFLOW] The file is a regular file, *nbytes* is greater than 0, the starting position is before
37303 the end-of-file, and the starting position is greater than or equal to the offset
37304 maximum established in the open file description associated with *fd*.

37305 The *read()* function shall fail if:

37306 [EAGAIN] or [EWOULDBLOCK]

37307 The file descriptor is for a socket, is marked O_NONBLOCK, and no data is
37308 waiting to be received.

37309 [ECONNRESET] A read was attempted on a socket and the connection was forcibly closed by
37310 its peer.

37311 [ENOTCONN] A read was attempted on a socket that is not connected.

37312 [ETIMEDOUT] A read was attempted on a socket and a transmission timeout occurred.

37313 XSI The *read()* and *pread()* functions may fail if:

37314 [EIO] A physical I/O error has occurred.

37315 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

37316 [ENOMEM] Insufficient memory was available to fulfill the request.

37317 [ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

37318

37319 The *pread()* function shall fail, and the file pointer shall remain unchanged, if:

37320 XSI [EINVAL] The *offset* argument is invalid. The value is negative.

37321 XSI [EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the file.

37322

37323 XSI [ENXIO] A request was outside the capabilities of the device.

37324 XSI [ESPIPE] *fildes* is associated with a pipe or FIFO.

37325 EXAMPLES

37326 Reading Data into a Buffer

37327 The following example reads data from the file associated with the file descriptor *fd* into the
37328 buffer pointed to by *buf*.

```
37329 #include <sys/types.h>
37330 #include <unistd.h>
37331 ...
37332 char buf[20];
37333 size_t nbytes;
37334 ssize_t bytes_read;
37335 int fd;
37336 ...
37337 nbytes = sizeof(buf);
37338 bytes_read = read(fd, buf, nbytes);
37339 ...
```

37340 APPLICATION USAGE

37341 None.

37342 RATIONALE

37343 This volume of IEEE Std 1003.1-2001 does not specify the value of the file offset after an error is
37344 returned; there are too many cases. For programming errors, such as [EBADF], the concept is
37345 meaningless since no file is involved. For errors that are detected immediately, such as
37346 [EAGAIN], clearly the pointer should not change. After an interrupt or hardware error, however,
37347 an updated value would be very useful and is the behavior of many implementations.

37348 Note that a *read()* of zero bytes does not modify *st_atime*. A *read()* that requests more than zero
37349 bytes, but returns zero, shall modify *st_atime*.

37350 Implementations are allowed, but not required, to perform error checking for *read()* requests of
37351 zero bytes.

37352

Input and Output

37353

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.

37354

37355

37356

37357

37358

37359

37360

37361

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.

37362

37363

37364

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.

37365

37366

37367

37368

37369

37370

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).

37371

37372

37373

37374

37375

37376

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.

37377

37378

37379

37380

37381

37382

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.

37383

37384

37385

37386

37387

37388

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.

37389

37390

37391

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.

37392

37393

37394

37395

37396

37397

37398

37399

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

37400 which *read()* and *write()* return the number of bytes actually transferred before the interrupt. If
37401 -1 is returned when any data is transferred, it is difficult to recover from the error on a seekable
37402 device and impossible on a non-seekable device. Most new implementations support this
37403 behavior. The behavior required by IEEE Std 1003.1-2001 is to return the number of bytes
37404 transferred.

37405 IEEE Std 1003.1-2001 does not specify when an implementation that buffers *read()*'s actually 2
37406 moves the data into the user-supplied buffer, so an implementation may choose to do this at the
37407 latest possible moment. Therefore, an interrupt arriving earlier may not cause *read()* to return a
37408 partial byte count, but rather to return -1 and set *errno* to [EINTR].

37409 Consideration was also given to combining the two previous options, and setting *errno* to
37410 [EINTR] while returning a short count. However, not only is there no existing practice that
37411 implements this, it is also contradictory to the idea that when *errno* is set, the function
37412 responsible shall return -1.

37413 FUTURE DIRECTIONS

37414 None.

37415 SEE ALSO

37416 *fcntl()*, *ioctl()*, *lseek()*, *open()*, *pipe()*, *readv()*, the Base Definitions volume of
37417 IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <stropts.h>, <sys/uio.h>,
37418 <unistd.h>

37419 CHANGE HISTORY

37420 First released in Issue 1. Derived from Issue 1 of the SVID.

37421 Issue 5

37422 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
37423 Threads Extension.

37424 Large File Summit extensions are added.

37425 The *pread()* function is added.

37426 Issue 6

37427 The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are
37428 marked as part of the XSI STREAMS Option Group.

37429 The following new requirements on POSIX implementations derive from alignment with the
37430 Single UNIX Specification:

37431 • The DESCRIPTION now states that if *read()* is interrupted by a signal after it has successfully
37432 read some data, it returns the number of bytes read. In Issue 3, it was optional whether *read()*
37433 returned the number of bytes read, or whether it returned -1 with *errno* set to [EINTR]. This
37434 is a FIPS requirement.

37435 • In the DESCRIPTION, text is added to indicate that for regular files, no data transfer occurs
37436 past the offset maximum established in the open file description associated with *fd*. This
37437 change is to support large files.

37438 • The [EOVERFLOW] mandatory error condition is added.

37439 • The [ENXIO] optional error condition is added.

37440 Text referring to sockets is added to the DESCRIPTION.

37441 The following changes were made to align with the IEEE P1003.1a draft standard:

37442 • The effect of reading zero bytes is clarified.

37443	The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that	
37444	read() results are unspecified for typed memory objects.	
37445	New RATIONALE is added to explain the atomicity requirements for input and output	
37446	operations.	
37447	The following error conditions are added for operations on sockets: [EAGAIN],	
37448	[ECONNRESET], [ENOTCONN], and [ETIMEDOUT].	
37449	The [EIO] error is made optional.	
37450	The following error conditions are added for operations on sockets: [ENOBUFS] and	
37451	[ENOMEM].	
37452	The <i>readv()</i> function is split out into a separate reference page.	
37453	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/108 is applied, updating the [EAGAIN]	2
37454	error in the ERRORS section from “the process would be delayed” to “the thread would be	2
37455	delayed”.	2
37456	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/109 is applied, making an editorial	2
37457	correction in the RATIONALE section.	2

37458 NAME

37459 readdir, readdir_r — read a directory

37460 SYNOPSIS

```
37461     #include <dirent.h>
37462
37462     struct dirent *readdir(DIR *dirp);
37463 TSF     int readdir_r(DIR *restrict dirp, struct dirent *restrict entry,
37464             struct dirent **restrict result);
```

37466 DESCRIPTION

37467 The type **DIR**, which is defined in the **<dirent.h>** header, represents a *directory stream*, which is
 37468 an ordered sequence of all the directory entries in a particular directory. Directory entries
 37469 represent files; files may be removed from a directory or added to a directory asynchronously to
 37470 the operation of *readdir()*.

37471 The *readdir()* function shall return a pointer to a structure representing the directory entry at the
 37472 current position in the directory stream specified by the argument *dirp*, and position the
 37473 directory stream at the next entry. It shall return a null pointer upon reaching the end of the
 37474 directory stream. The structure **dirent** defined in the **<dirent.h>** header describes a directory
 37475 entry.

37476 The *readdir()* function shall not return directory entries containing empty names. If entries for
 37477 dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-
 37478 dot; otherwise, they shall not be returned.

37479 The pointer returned by *readdir()* points to data which may be overwritten by another call to
 37480 *readdir()* on the same directory stream. This data is not overwritten by another call to *readdir()*
 37481 on a different directory stream.

37482 If a file is removed from or added to the directory after the most recent call to *opendir()* or
 37483 *rewinddir()*, whether a subsequent call to *readdir()* returns an entry for that file is unspecified.

37484 The *readdir()* function may buffer several directory entries per actual read operation; *readdir()*
 37485 shall mark for update the *st_atime* field of the directory each time the directory is actually read.

37486 After a call to *fork()*, either the parent or child (but not both) may continue processing the
 37487 XSI directory stream using *readdir()*, *rewinddir()*, or *seekdir()*. If both the parent and child processes
 37488 use these functions, the result is undefined.

37489 If the entry names a symbolic link, the value of the *d_ino* member is unspecified.

37490 The *readdir()* function need not be reentrant. A function that is not required to be reentrant is not
 37491 required to be thread-safe.

37492 TSF The *readdir_r()* function shall initialize the **dirent** structure referenced by *entry* to represent the
 37493 directory entry at the current position in the directory stream referred to by *dirp*, store a pointer
 37494 to this structure at the location referenced by *result*, and position the directory stream at the next
 37495 entry.

37496 The storage pointed to by *entry* shall be large enough for a **dirent** with an array of **char** *d_name*
 37497 members containing at least {NAME_MAX}+1 elements.

37498 Upon successful return, the pointer returned at **result* shall have the same value as the argument
 37499 *entry*. Upon reaching the end of the directory stream, this pointer shall have the value NULL.

37500 The *readdir_r()* function shall not return directory entries containing empty names.

37501 If a file is removed from or added to the directory after the most recent call to *opendir()* or
 37502 *rewinddir()*, whether a subsequent call to *readdir_r()* returns an entry for that file is unspecified.

37503 The *readdir_r()* function may buffer several directory entries per actual read operation; the
 37504 *readdir_r()* function shall mark for update the *st_atime* field of the directory each time the
 37505 directory is actually read.

37506 Applications wishing to check for error situations should set *errno* to 0 before calling *readdir()*. If
 37507 *errno* is set to non-zero on return, an error occurred.

37508 RETURN VALUE

37509 Upon successful completion, *readdir()* shall return a pointer to an object of type **struct dirent**.
 37510 When an error is encountered, a null pointer shall be returned and *errno* shall be set to indicate
 37511 the error. When the end of the directory is encountered, a null pointer shall be returned and *errno*
 37512 is not changed.

37513 TSF If successful, the *readdir_r()* function shall return zero; otherwise, an error number shall be
 37514 returned to indicate the error.

37515 ERRORS

37516 The *readdir()* function shall fail if:

37517 [EOVERFLOW] One of the values in the structure to be returned cannot be represented
 37518 correctly.

37519 The *readdir()* function may fail if:

37520 [EBADF] The *dirp* argument does not refer to an open directory stream.

37521 [ENOENT] The current position of the directory stream is invalid.

37522 The *readdir_r()* function may fail if:

37523 [EBADF] The *dirp* argument does not refer to an open directory stream.

37524 EXAMPLES

37525	The following sample program searches the current directory for each of the arguments supplied	1
37526	on the command line.	1
37527	#include <dirent.h>	1
37528	#include <errno.h>	1
37529	#include <stdio.h>	1
37530	#include <string.h>	1
37531	static void lookup(const char *arg)	1
37532	{	1
37533	DIR *dirp;	1
37534	struct dirent *dp;	1
37535	if ((dirp = opendir(".")) == NULL) {	1
37536	perror("couldn't open '.'");	1
37537	return;	1
37538	}	1
37539	do {	1
37540	errno = 0;	1
37541	if ((dp = readdir(dirp)) != NULL) {	1
37542	if (strcmp(dp->d_name, arg) != 0)	1
37543	continue;	1

```

37544         (void) printf("found %s\n", arg);
37545         (void) closedir(dirp);
37546         return;
37547     }
37548 } while (dp != NULL);
37549 if (errno != 0)
37550     perror("error reading directory");
37551 else
37552     (void) printf("failed to find %s\n", arg);
37553     (void) closedir(dirp);
37554     return;
37555 }
37556 int main(int argc, char *argv[])
37557 {
37558     int i;
37559     for (i = 1; i < argc; i++)
37560         lookup(argv[i]);
37561     return (0);
37562 }
```

37563 APPLICATION USAGE

The *readdir()* function should be used in conjunction with *opendir()*, *closedir()*, and *rewinddir()* to examine the contents of the directory.

The *readdir_r()* function is thread-safe and shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

37568 RATIONALE

The returned value of *readdir()* merely *represents* a directory entry. No equivalence should be inferred.

Historical implementations of *readdir()* obtain multiple directory entries on a single read operation, which permits subsequent *readdir()* operations to operate from the buffered information. Any wording that required each successful *readdir()* operation to mark the directory *st_atime* field for update would disallow such historical performance-oriented implementations.

Since *readdir()* returns NULL when it detects an error and when the end of the directory is encountered, an application that needs to tell the difference must set *errno* to zero before the call and check it if NULL is returned. Since the function must not change *errno* in the second case and must set it to a non-zero value in the first case, a zero *errno* after a call returning NULL indicates end-of-directory; otherwise, an error.

Routines to deal with this problem more directly were proposed:

```

37582 int derror (dirp)
37583 DIR *dirp;
37584 void clearderr (dirp)
37585 DIR *dirp;
```

The first would indicate whether an error had occurred, and the second would clear the error indication. The simpler method involving *errno* was adopted instead by requiring that *readdir()* not change *errno* when end-of-directory is encountered.

37589 An error or signal indicating that a directory has changed while open was considered but
37590 rejected.

37591 The thread-safe version of the directory reading function returns values in a user-supplied buffer
37592 instead of possibly using a static data area that may be overwritten by each call. Either the
37593 {NAME_MAX} compile-time constant or the corresponding *pathconf()* option can be used to
37594 determine the maximum sizes of returned pathnames.

37595 FUTURE DIRECTIONS

37596 None.

37597 SEE ALSO

37598 *closedir()*, *Istat()*, *opendir()*, *rewinddir()*, *symlink()*, the Base Definitions volume of
37599 IEEE Std 1003.1-2001, <dirent.h>, <sys/types.h>

37600 CHANGE HISTORY

37601 First released in Issue 2.

37602 Issue 5

37603 Large File Summit extensions are added.

37604 The *readdir_r()* function is included for alignment with the POSIX Threads Extension.

37605 A note indicating that the *readdir()* function need not be reentrant is added to the
37606 DESCRIPTION.

37607 Issue 6

37608 The *readdir_r()* function is marked as part of the Thread-Safe Functions option.

37609 The Open Group Corrigendum U026/7 is applied, correcting the prototype for *readdir_r()*.

37610 The Open Group Corrigendum U026/8 is applied, clarifying the wording of the successful
37611 return for the *readdir_r()* function.

37612 The following new requirements on POSIX implementations derive from alignment with the
37613 Single UNIX Specification:

- 37614 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
37615 required for conforming implementations of previous POSIX specifications, it was not
37616 required for UNIX applications.
- 37617 • A statement is added to the DESCRIPTION indicating the disposition of certain fields in
37618 **struct dirent** when an entry refers to a symbolic link.
- 37619 • The [EOVERFLOW] mandatory error condition is added. This change is to support large
37620 files.
- 37621 • The [ENOENT] optional error condition is added.

37622 The APPLICATION USAGE section is updated to include a note on the thread-safe function and
37623 its avoidance of possibly using a static data area.

37624 The **restrict** keyword is added to the *readdir_r()* prototype for alignment with the
37625 ISO/IEC 9899: 1999 standard.

37626 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/50 is applied, replacing the EXAMPLES 1
37627 section with a new example.

37628 NAME

37629 readlink — read the contents of a symbolic link

37630 SYNOPSIS

```
37631        #include <unistd.h>
37632        ssize_t readlink(const char *restrict path, char *restrict buf,
37633                    size_t bufsize);
```

37634 DESCRIPTION

37635 The *readlink()* function shall place the contents of the symbolic link referred to by *path* in the
37636 buffer *buf* which has size *bufsize*. If the number of bytes in the symbolic link is less than *bufsize*,
37637 the contents of the remainder of *buf* are unspecified. If the *buf* argument is not large enough to
37638 contain the link content, the first *bufsize* bytes shall be placed in *buf*.

37639 If the value of *bufsize* is greater than {SSIZE_MAX}, the result is implementation-defined.

37640 RETURN VALUE

37641 Upon successful completion, *readlink()* shall return the count of bytes placed in the buffer.
37642 Otherwise, it shall return a value of -1, leave the buffer unchanged, and set *errno* to indicate the
37643 error.

37644 ERRORS

37645 The *readlink()* function shall fail if:

37646 [EACCES] Search permission is denied for a component of the path prefix of *path*.
37647 [EINVAL] The *path* argument names a file that is not a symbolic link.
37648 [EIO] An I/O error occurred while reading from the file system.
37649 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
37650 argument.
37651 [ENAMETOOLONG]
37652 The length of the *path* argument exceeds {PATH_MAX} or a pathname
37653 component is longer than {NAME_MAX}.

37654 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

37655 [ENOTDIR] A component of the path prefix is not a directory.

37656 The *readlink()* function may fail if:

37657 [EACCES] Read permission is denied for the directory.

37658 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
37659 resolution of the *path* argument.

37660 [ENAMETOOLONG]

37661 As a result of encountering a symbolic link in resolution of the *path* argument,
37662 the length of the substituted pathname string exceeded {PATH_MAX}.

37663 EXAMPLES

37664 **Reading the Name of a Symbolic Link**

37665 The following example shows how to read the name of a symbolic link named /modules/pass1.

```
37666 #include <unistd.h>
37667 char buf[1024];
37668 ssize_t len;
37669 ...
37670 if ((len = readlink("/modules/pass1", buf, sizeof(buf)-1)) != -1)
37671     buf[len] = '\0';
```

37672 APPLICATION USAGE

37673 Conforming applications should not assume that the returned contents of the symbolic link are
37674 null-terminated.

37675 RATIONALE

37676 Since IEEE Std 1003.1-2001 does not require any association of file times with symbolic links,
37677 there is no requirement that file times be updated by *readlink()*. The type associated with *bufsiz*
37678 is a *size_t* in order to be consistent with both the ISO C standard and the definition of *read()*.
37679 The behavior specified for *readlink()* when *bufsiz* is zero represents historical practice. For this
37680 case, the standard developers considered a change whereby *readlink()* would return the number
37681 of non-null bytes contained in the symbolic link with the buffer *buf* remaining unchanged;
37682 however, since the *stat* structure member *st_size* value can be used to determine the size of
37683 buffer necessary to contain the contents of the symbolic link as returned by *readlink()*, this
37684 proposal was rejected, and the historical practice retained.

37685 FUTURE DIRECTIONS

37686 None.

37687 SEE ALSO

37688 *lstat()*, *stat()*, *symlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

37689 CHANGE HISTORY

37690 First released in Issue 4, Version 2.

37691 Issue 5

37692 Moved from X/OPEN UNIX extension to BASE.

37693 Issue 6

37694 The return type is changed to *ssize_t*, to align with the IEEE P1003.1a draft standard.

37695 The following new requirements on POSIX implementations derive from alignment with the
37696 Single UNIX Specification:

- 37697 • This function is made mandatory.
- 37698 • In this function it is possible for the return value to exceed the range of the type *ssize_t* (since
37699 *size_t* has a larger range of positive values than *ssize_t*). A sentence restricting the size of
37700 the *size_t* object is added to the description to resolve this conflict.

37701 The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- 37702 • The FUTURE DIRECTIONS section is changed to None.

37703 The following changes were made to align with the IEEE P1003.1a draft standard:

- 37704 • The [ELOOP] optional error condition is added.

37705
37706

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

37707 NAME

37708 readv — read a vector

37709 SYNOPSIS

37710 XSI #include <sys/uio.h>

```
37711        ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);
```

37712

37713 DESCRIPTION

37714 The *readv()* function shall be equivalent to *read()*, except as described below. The *readv()* function shall place the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov[0]*, *iov[1]*, ..., *iov[iovcnt-1]*. The *iovcnt* argument is valid if greater than 0 and less than or equal to {IOV_MAX}.

37718 Each *iovec* entry specifies the base address and length of an area in memory where data should
37719 be placed. The *readv()* function shall always fill an area completely before proceeding to the
37720 next.

37721 Upon successful completion, *readv()* shall mark for update the *st_atime* field of the file.

37722 RETURN VALUE

37723 Refer to *read()*.

37724 ERRORS

37725 Refer to *read()*.

37726 In addition, the *readv()* function shall fail if:

37727 [EINVAL] The sum of the *iov_len* values in the *iov* array overflowed an *ssize_t*.

37728 The *readv()* function may fail if:

37729 [EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.

37730 EXAMPLES

37731 **Reading Data into an Array**

37732 The following example reads data from the file associated with the file descriptor *fd* into the
37733 buffers specified by members of the *iov* array.

```
37734        #include <sys/types.h>
37735        #include <sys/uio.h>
37736        #include <unistd.h>
37737        ...
37738        ssize_t bytes_read;
37739        int fd;
37740        char buf0[20];
37741        char buf1[30];
37742        char buf2[40];
37743        int iovcnt;
37744        struct iovec iov[3];

37745        iov[0].iov_base = buf0;
37746        iov[0].iov_len = sizeof(buf0);
37747        iov[1].iov_base = buf1;
37748        iov[1].iov_len = sizeof(buf1);
37749        iov[2].iov_base = buf2;
37750        iov[2].iov_len = sizeof(buf2);
```

```
37751     ...
37752     iovcnt = sizeof(iov) / sizeof(struct iovec);
37753     bytes_read = readv(fd, iov, iovcnt);
37754     ...
37755 APPLICATION USAGE
37756     None.
37757 RATIONALE
37758     Refer to read().
37759 FUTURE DIRECTIONS
37760     None.
37761 SEE ALSO
37762     read(), writev(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/uio.h>
37763 CHANGE HISTORY
37764     First released in Issue 4, Version 2.
37765 Issue 6
37766     Split out from the read() reference page.
```

37767 NAME

37768 realloc — memory reallocator

37769 SYNOPSIS

```
37770       #include <stdlib.h>
37771       void *realloc(void *ptr, size_t size);
```

37772 DESCRIPTION

37773 CX The functionality described on this reference page is aligned with the ISO C standard. Any
37774 conflict between the requirements described here and the ISO C standard is unintentional. This
37775 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

37776 The *realloc()* function shall change the size of the memory object pointed to by *ptr* to the size
37777 specified by *size*. The contents of the object shall remain unchanged up to the lesser of the new
37778 and old sizes. If the new size of the memory object would require movement of the object, the
37779 space for the previous instantiation of the object is freed. If the new size is larger, the contents of
37780 the newly allocated portion of the object are unspecified. If *size* is 0 and *ptr* is not a null pointer,
37781 the object pointed to is freed. If the space cannot be allocated, the object shall remain unchanged.

37782 If *ptr* is a null pointer, *realloc()* shall be equivalent to *malloc()* for the specified size.

37783 If *ptr* does not match a pointer returned earlier by *calloc()*, *malloc()*, or *realloc()* or if the space has
37784 previously been deallocated by a call to *free()* or *realloc()*, the behavior is undefined.

37785 The order and contiguity of storage allocated by successive calls to *realloc()* is unspecified. The
37786 pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to
37787 a pointer to any type of object and then used to access such an object in the space allocated (until
37788 the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object
37789 disjoint from any other object. The pointer returned shall point to the start (lowest byte address)
37790 of the allocated space. If the space cannot be allocated, a null pointer shall be returned.

37791 RETURN VALUE

37792 Upon successful completion with a *size* not equal to 0, *realloc()* shall return a pointer to the
37793 (possibly moved) allocated space. If *size* is 0, either a null pointer or a unique pointer that can be
37794 successfully passed to *free()* shall be returned. If there is not enough available memory, *realloc()*
37795 CX shall return a null pointer and set *errno* to [ENOMEM].

37796 ERRORS

37797 The *realloc()* function shall fail if:

37798 CX [ENOMEM] Insufficient memory is available.

37799 EXAMPLES

37800 None.

37801 APPLICATION USAGE

37802 None.

37803 RATIONALE

37804 None.

37805 FUTURE DIRECTIONS

37806 None.

37807 SEE ALSO

37808 *calloc()*, *free()*, *malloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

37809 CHANGE HISTORY

37810 First released in Issue 1. Derived from Issue 1 of the SVID.

37811 Issue 6

37812 Extensions beyond the ISO C standard are marked.

37813 The following new requirements on POSIX implementations derive from alignment with the
37814 Single UNIX Specification:

- 37815 • In the RETURN VALUE section, if there is not enough available memory, the setting of *errno*
37816 to [ENOMEM] is added.
- 37817 • The [ENOMEM] error condition is added.

37818 NAME

37819 realpath — resolve a pathname

37820 SYNOPSIS

37821 XSI #include <stdlib.h>
37822 char *realpath(const char *restrict file_name,
37823 char *restrict resolved_name);
37824

37825 DESCRIPTION

37826 The *realpath()* function shall derive, from the pathname pointed to by *file_name*, an absolute
37827 pathname that names the same file, whose resolution does not involve '.', '..', or symbolic
37828 links. The generated pathname shall be stored as a null-terminated string, up to a maximum of
37829 {PATH_MAX} bytes, in the buffer pointed to by *resolved_name*. 1

37830 If *resolved_name* is a null pointer, the behavior of *realpath()* is implementation-defined. 1

37831 RETURN VALUE

37832 Upon successful completion, *realpath()* shall return a pointer to the resolved name. Otherwise,
37833 *realpath()* shall return a null pointer and set *errno* to indicate the error, and the contents of the
37834 buffer pointed to by *resolved_name* are undefined.

37835 ERRORS

37836 The *realpath()* function shall fail if:

37837 [EACCES] Read or search permission was denied for a component of *file_name*.
37838 [EINVAL] The *file_name* argument is a null pointer. 1
37839 [EIO] An error occurred while reading from the file system.
37840 [ELOOP] A loop exists in symbolic links encountered during resolution of the *file_name* 2
37841 argument.

37842 [ENAMETOOLONG]
37843 The length of the *file_name* argument exceeds {PATH_MAX} or a pathname
37844 component is longer than {NAME_MAX}.

37845 [ENOENT] A component of *file_name* does not name an existing file or *file_name* points to
37846 an empty string.

37847 [ENOTDIR] A component of the path prefix is not a directory.

37848 The *realpath()* function may fail if:

37849 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
37850 resolution of the *file_name* argument. 2

37851 [ENAMETOOLONG]
37852 Pathname resolution of a symbolic link produced an intermediate result
37853 whose length exceeds {PATH_MAX}.

37854 [ENOMEM] Insufficient storage space is available.

37855 EXAMPLES**37856 Generating an Absolute Pathname**

37857 The following example generates an absolute pathname for the file identified by the *symlinkpath* argument. The generated pathname is stored in the *actualpath* array.

```
37859 #include <stdlib.h>
37860 ...
37861 char *symlinkpath = "/tmp/symlink/file";
37862 char actualpath [PATH_MAX+1];
37863 char *ptr;
37864 ptr = realpath(symlinkpath, actualpath);
```

37865 APPLICATION USAGE

37866 None.

37867 RATIONALE

37868 Since the maximum pathname length is arbitrary unless {PATH_MAX} is defined, an application 1
37869 generally cannot supply a *resolved_name* buffer with size {{PATH_MAX}+1}. 1

37870 FUTURE DIRECTIONS

37871 In the future, passing a null pointer to *realpath()* for the *resolved_name* argument may be defined 1
37872 to have *realpath()* allocate space for the generated pathname. 1

37873 SEE ALSO

37874 *getcwd()*, *sysconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

37875 CHANGE HISTORY

37876 First released in Issue 4, Version 2.

37877 Issue 5

37878 Moved from X/OPEN UNIX extension to BASE.

37879 Issue 6

37880 The **restrict** keyword is added to the *realpath()* prototype for alignment with the 1
37881 ISO/IEC 9899:1999 standard.

37882 The wording of the mandatory [ELOOP] error condition is updated, and a second optional 1
37883 [ELOOP] error condition is added.

37884 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/51 is applied, adding new text to the 1
37885 DESCRIPTION for the case when *resolved_name* is a null pointer, changing the [EINVAL] error 1
37886 text, adding text to the RATIONALE, and adding text to FUTURE DIRECTIONS. 1

37887 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/110 is applied, updating the ERRORS 2
37888 section to refer to the *file_name* argument, rather than a non-existent *path* argument. 2

37889 **NAME**

37890 recv — receive a message from a connected socket

37891 **SYNOPSIS**

```
37892       #include <sys/socket.h>
37893       ssize_t recv(int socket, void *buffer, size_t length, int flags);
```

37894 **DESCRIPTION**

37895 The *recv()* function shall receive a message from a connection-mode or connectionless-mode socket. It is normally used with connected sockets because it does not permit the application to retrieve the source address of received data.

37898 The *recv()* function takes the following arguments:

37899 <i>socket</i>	Specifies the socket file descriptor.
37900 <i>buffer</i>	Points to a buffer where the message should be stored.
37901 <i>length</i>	Specifies the length in bytes of the buffer pointed to by the <i>buffer</i> argument.
37902 <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:
37904 MSG_PEEK	Peeks at an incoming message. The data is treated as unread and the next <i>recv()</i> or similar function shall still return this data.
37906 MSG_OOB	Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
37908 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.

37914 The *recv()* function shall return the length of the message written to the buffer pointed to by the *buffer* argument. For message-based sockets, such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message shall be read in a single operation. If a message is too long to fit in the supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes shall be discarded. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes available, and no data shall be discarded.

37921 If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first message.

37923 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, *recv()* shall block until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, *recv()* shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

37927 **RETURN VALUE**

37928 Upon successful completion, *recv()* shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recv()* shall return 0. Otherwise, -1 shall be returned and *errno* set to indicate the error.

37931 ERRORS

- 37932 The *recv()* function shall fail if:
- 37933 [EAGAIN] or [EWOULDBLOCK]
37934 The socket's file descriptor is marked O_NONBLOCK and no data is waiting
37935 to be received; or MSG_OOB is set and no out-of-band data is available and
37936 either the socket's file descriptor is marked O_NONBLOCK or the socket does
37937 not support blocking to await out-of-band data.
- 37938 [EBADF] The *socket* argument is not a valid file descriptor.
- 37939 [ECONNRESET] A connection was forcibly closed by a peer.
- 37940 [EINTR] The *recv()* function was interrupted by a signal that was caught, before any
37941 data was available.
- 37942 [EINVAL] The MSG_OOB flag is set and no out-of-band data is available.
- 37943 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.
- 37944 [ENOTSOCK] The *socket* argument does not refer to a socket.
- 37945 [EOPNOTSUPP] The specified flags are not supported for this socket type or protocol.
- 37946 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
37947 transmission timeout on active connection.
- 37948 The *recv()* function may fail if:
- 37949 [EIO] An I/O error occurred while reading from or writing to the file system.
- 37950 [ENOBUFS] Insufficient resources were available in the system to perform the operation.
- 37951 [ENOMEM] Insufficient memory was available to fulfill the request.

37952 EXAMPLES

- 37953 None.

37954 APPLICATION USAGE

- 37955 The *recv()* function is equivalent to *recvfrom()* with a zero *address_len* argument, and to *read()* if
37956 no flags are used.
- 37957 The *select()* and *poll()* functions can be used to determine when data is available to be received.

37958 RATIONALE

- 37959 None.

37960 FUTURE DIRECTIONS

- 37961 None.

37962 SEE ALSO

- 37963 *poll()*, *read()*, *recvmsg()*, *recvfrom()*, *select()*, *send()*, *sendmsg()*, *sendto()*, *shutdown()*, *socket()*,
37964 *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

37965 CHANGE HISTORY

- 37966 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

37967 NAME

37968 recvfrom — receive a message from a socket

37969 SYNOPSIS

```
37970       #include <sys/socket.h>
37971       ssize_t recvfrom(int socket, void *restrict buffer, size_t length,
37972            int flags, struct sockaddr *restrict address,
37973            socklen_t *restrict address_len);
```

37974 DESCRIPTION

37975 The *recvfrom()* function shall receive a message from a connection-mode or connectionless-mode
37976 socket. It is normally used with connectionless-mode sockets because it permits the application
37977 to retrieve the source address of received data.

37978 The *recvfrom()* function takes the following arguments:

37979 <i>socket</i>	Specifies the socket file descriptor.		
37980 <i>buffer</i>	Points to the buffer where the message should be stored.		
37981 <i>length</i>	Specifies the length in bytes of the buffer pointed to by the <i>buffer</i> argument.		
37982 <i>flags</i>	Specifies the type of message reception. Values of this argument are formed 37983 by logically OR'ing zero or more of the following values: 37984 MSG_PEEK Peeks at an incoming message. The data is treated as unread 37985 and the next <i>recvfrom()</i> or similar function shall still return 37986 this data. 37987 MSG_OOB Requests out-of-band data. The significance and semantics 37988 of out-of-band data are protocol-specific. 37989 MSG_WAITALL On SOCK_STREAM sockets this requests that the function 37990 block until the full amount of data can be returned. The 37991 function may return the smaller amount of data if the socket 37992 is a message-based socket, if a signal is caught, if the 37993 connection is terminated, if MSG_PEEK was specified, or if 37994 an error is pending for the socket. 37995 <i>address</i>	A null pointer, or points to a sockaddr structure in which the sending address 37996 is to be stored. The length and format of the address depend on the address 37997 family of the socket. 37998 <i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> 37999 argument.

38000 The *recvfrom()* function shall return the length of the message written to the buffer pointed to by
38001 RS the *buffer* argument. For message-based sockets, such as **SOCK_RAW**, **SOCK_DGRAM**, and
38002 **SOCK_SEQPACKET**, the entire message shall be read in a single operation. If a message is too
38003 long to fit in the supplied buffer, and **MSG_PEEK** is not set in the *flags* argument, the excess
38004 bytes shall be discarded. For stream-based sockets, such as **SOCK_STREAM**, message
38005 boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes
38006 available, and no data shall be discarded.

38007 If the **MSG_WAITALL** flag is not set, data shall be returned only up to the end of the first
38008 message.

38009 Not all protocols provide the source address for messages. If the *address* argument is not a null
38010 pointer and the protocol provides the source address of messages, the source address of the

38011 received message shall be stored in the **sockaddr** structure pointed to by the *address* argument,
38012 and the length of this address shall be stored in the object pointed to by the *address_len*
38013 argument.

38014 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
38015 the stored address shall be truncated.

38016 If the *address* argument is not a null pointer and the protocol does not provide the source address
38017 of messages, the value stored in the object pointed to by *address* is unspecified.

38018 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
38019 descriptor, *recvfrom()* shall block until a message arrives. If no messages are available at the
38020 socket and O_NONBLOCK is set on the socket's file descriptor, *recvfrom()* shall fail and set *errno*
38021 to [EAGAIN] or [EWOULDBLOCK].

38022 RETURN VALUE

38023 Upon successful completion, *recvfrom()* shall return the length of the message in bytes. If no
38024 messages are available to be received and the peer has performed an orderly shutdown,
38025 *recvfrom()* shall return 0. Otherwise, the function shall return -1 and set *errno* to indicate the
38026 error.

38027 ERRORS

38028 The *recvfrom()* function shall fail if:

38029 [EAGAIN] or [EWOULDBLOCK]

38030 The socket's file descriptor is marked O_NONBLOCK and no data is waiting
38031 to be received; or MSG_OOB is set and no out-of-band data is available and
38032 either the socket's file descriptor is marked O_NONBLOCK or the socket does
38033 not support blocking to await out-of-band data.

38034 [EBADF] The *socket* argument is not a valid file descriptor.

38035 [ECONNRESET] A connection was forcibly closed by a peer.

38036 [EINTR] A signal interrupted *recvfrom()* before any data was available.

38037 [EINVAL] The MSG_OOB flag is set and no out-of-band data is available.

38038 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

38039 [ENOTSOCK] The *socket* argument does not refer to a socket.

38040 [EOPNOTSUPP] The specified flags are not supported for this socket type.

38041 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
38042 transmission timeout on active connection.

38043 The *recvfrom()* function may fail if:

38044 [EIO] An I/O error occurred while reading from or writing to the file system.

38045 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

38046 [ENOMEM] Insufficient memory was available to fulfill the request.

38047 EXAMPLES

38048 None.

38049 APPLICATION USAGE

38050 The *select()* and *poll()* functions can be used to determine when data is available to be received.

38051 RATIONALE

38052 None.

38053 FUTURE DIRECTIONS

38054 None.

38055 SEE ALSO

38056 *poll()*, *read()*, *recv()*, *recvmsg()*, *select()*, *send()*, *sendmsg()*, *sendto()*, *shutdown()*, *socket()*, *write()*,
38057 the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

38058 CHANGE HISTORY

38059 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

38060 NAME

38061 recvmsg — receive a message from a socket

38062 SYNOPSIS

```
38063       #include <sys/socket.h>
38064       ssize_t recvmsg(int socket, struct msghdr *message, int flags);
```

38065 DESCRIPTION

38066 The *recvmsg()* function shall receive a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.

38069 The *recvmsg()* function takes the following arguments:

38070 *socket* Specifies the socket file descriptor.

38071 *message* 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 *msg_flags* member is ignored on input, but may contain meaningful values on output.

38075 *flags* Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

38077 MSG_OOB Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.

38079 MSG_PEEK Peeks at the incoming message.

38080 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.

38086 The *recvmsg()* function shall receive messages from unconnected or connected sockets and shall return the length of the message.

38088 The *recvmsg()* function shall return the total length of the message. For message-based sockets, such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message shall be read in a single operation. If a message is too long to fit in the supplied buffers, and MSG_PEEK is not set in the *flags* argument, the excess bytes shall be discarded, and MSG_TRUNC shall be set in the *msg_flags* member of the **msghdr** structure. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes available, and no data shall be discarded.

38095 If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first message.

38097 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, *recvmsg()* shall block until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, the *recvmsg()* function shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

38101 In the **msghdr** structure, the *msg_name* and *msg_namelen* members specify the source address if the socket is unconnected. If the socket is connected, the *msg_name* and *msg_namelen* members shall be ignored. The *msg_name* member may be a null pointer if no names are desired or required. The *msg iov* and *msg_ivolen* fields are used to specify where the received data shall be

38105 stored. *msg iov* points to an array of **iovec** structures; *msg iovlen* shall be set to the dimension of
38106 this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field
38107 gives its size in bytes. Each storage area indicated by *msg iov* is filled with received data in turn
38108 until all of the received data is stored or all of the areas have been filled.

38109 Upon successful completion, the *msg_flags* member of the message header shall be the bitwise-
38110 inclusive OR of all of the following flags that indicate conditions detected for the received
38111 message:

38112	MSG_EOR	End-of-record was received (if supported by the protocol).
38113	MSG_OOB	Out-of-band data was received.
38114	MSG_TRUNC	Normal data was truncated.
38115	MSG_CTRUNC	Control data was truncated.

38116 RETURN VALUE

38117 Upon successful completion, *recvmsg()* shall return the length of the message in bytes. If no
38118 messages are available to be received and the peer has performed an orderly shutdown,
38119 *recvmsg()* shall return 0. Otherwise, -1 shall be returned and *errno* set to indicate the error.

38120 ERRORS

38121 The *recvmsg()* function shall fail if:

38122 [EAGAIN] or [EWOULDBLOCK]

38123 The socket's file descriptor is marked O_NONBLOCK and no data is waiting
38124 to be received; or MSG_OOB is set and no out-of-band data is available and
38125 either the socket's file descriptor is marked O_NONBLOCK or the socket does
38126 not support blocking to await out-of-band data.

38127 [EBADF] The *socket* argument is not a valid open file descriptor.

38128 [ECONNRESET] A connection was forcibly closed by a peer.

38129 [EINTR] This function was interrupted by a signal before any data was available.

38130 [EINVAL] The sum of the *iov_len* values overflows a **ssize_t**, or the MSG_OOB flag is set
38131 and no out-of-band data is available.

38132 [EMSGSIZE] The *msg iovlen* member of the **msghdr** structure pointed to by *message* is less
38133 than or equal to 0, or is greater than {IOV_MAX}.

38134 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

38135 [ENOTSOCK] The *socket* argument does not refer to a socket.

38136 [EOPNOTSUPP] The specified flags are not supported for this socket type.

38137 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
38138 transmission timeout on active connection.

38139 The *recvmsg()* function may fail if:

38140 [EIO] An I/O error occurred while reading from or writing to the file system.

38141 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

38142 [ENOMEM] Insufficient memory was available to fulfill the request.

38143 EXAMPLES

38144 None.

38145 APPLICATION USAGE

38146 The *select()* and *poll()* functions can be used to determine when data is available to be received.

38147 RATIONALE

38148 None.

38149 FUTURE DIRECTIONS

38150 None.

38151 SEE ALSO

38152 *poll()*, *recv()*, *recvfrom()*, *select()*, *send()*, *sendmsg()*, *sendto()*, *shutdown()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

38154 CHANGE HISTORY

38155 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

38156 NAME

38157 regcomp, regerror, regexec, regfree — regular expression matching

38158 SYNOPSIS

```
38159     #include <regex.h>
38160
38161     int regcomp(regex_t *restrict preg, const char *restrict pattern,
38162                 int cflags);
38163     size_t regerror(int errcode, const regex_t *restrict preg,
38164                     char *restrict errbuf, size_t errbuf_size);
38165     int regexec(const regex_t *restrict preg, const char *restrict string,
38166                  size_t nmatch, regmatch_t pmatch[restrict], int eflags);
38167     void regfree(regex_t *preg);
```

38167 DESCRIPTION

38168 These functions interpret *basic* and *extended* regular expressions as described in the Base
38169 Definitions volume of IEEE Std 1003.1-2001, Chapter 9, Regular Expressions.

38170 The **regex_t** structure is defined in **<regex.h>** and contains at least the following member:

Member Type	Member Name	Description
size_t	re_nsub	Number of parenthesized subexpressions.

38174 The **regmatch_t** structure is defined in **<regex.h>** and contains at least the following members:

Member Type	Member Name	Description
regoff_t	<i>rm_so</i>	Byte offset from start of <i>string</i> to start of substring.
regoff_t	<i>rm_eo</i>	Byte offset from start of <i>string</i> of the first character after the end of substring.

38180 The *regcomp()* function shall compile the regular expression contained in the string pointed to by
38181 the *pattern* argument and place the results in the structure pointed to by *preg*. The *cflags* argument
38182 is the bitwise-inclusive OR of zero or more of the following flags, which are defined in
38183 the **<regex.h>** header:

38184 REG_EXTENDED Use Extended Regular Expressions.
38185 REG_ICASE Ignore case in match. (See the Base Definitions volume of
38186 IEEE Std 1003.1-2001, Chapter 9, Regular Expressions.)

38187 REG_NOSUB Report only success/fail in *regexec()*.
38188 REG_NEWLINE Change the handling of **<newline>**s, as described in the text.
38189 The default regular expression type for *pattern* is a Basic Regular Expression. The application can
38190 specify Extended Regular Expressions using the REG_EXTENDED *cflags* flag.

38191 If the REG_NOSUB flag was not set in *cflags*, then *regcomp()* shall set *re_nsub* to the number of
38192 parenthesized subexpressions (delimited by "*\(\)*" in basic regular expressions or "*()*" in
38193 extended regular expressions) found in *pattern*.

38194 The *regexec()* function compares the null-terminated string specified by *string* with the compiled
38195 regular expression *preg* initialized by a previous call to *regcomp()*. If it finds a match, *regexec()*
38196 shall return 0; otherwise, it shall return non-zero indicating either no match or an error. The
38197 *eflags* argument is the bitwise-inclusive OR of zero or more of the following flags, which are
38198 defined in the **<regex.h>** header:

38199 REG_NOTEOL The first character of the string pointed to by *string* is not the beginning of the
 38200 line. Therefore, the circumflex character ('^'), when taken as a special
 38201 character, shall not match the beginning of *string*.

38202 REG_NOTEOL The last character of the string pointed to by *string* is not the end of the line.
 38203 Therefore, the dollar sign ('\$'), when taken as a special character, shall not
 38204 match the end of *string*.

38205 If *nmatch* is 0 or REG_NOSUB was set in the *cflags* argument to *regcomp()*, then *regexec()* shall
 38206 ignore the *pmatch* argument. Otherwise, the application shall ensure that the *pmatch* argument
 38207 points to an array with at least *nmatch* elements, and *regexec()* shall fill in the elements of that
 38208 array with offsets of the substrings of *string* that correspond to the parenthesized subexpressions
 38209 of *pattern*: *pmatch*[*i*].*rm_so* shall be the byte offset of the beginning and *pmatch*[*i*].*rm_eo* shall be
 38210 one greater than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th
 38211 matched open parenthesis, counting from 1.) Offsets in *pmatch*[0] identify the substring that
 38212 corresponds to the entire regular expression. Unused elements of *pmatch* up to *pmatch*[*nmatch*-1]
 38213 shall be filled with -1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself
 38214 counts as a subexpression), then *regexec()* shall still do the match, but shall record only the first
 38215 *nmatch* substrings.

38216 When matching a basic or extended regular expression, any given parenthesized subexpression
 38217 of *pattern* might participate in the match of several different substrings of *string*, or it might not
 38218 match any substring even though the pattern as a whole did match. The following rules shall be
 38219 used to determine which substrings to report in *pmatch* when matching regular expressions:

38220 1. If subexpression *i* in a regular expression is not contained within another subexpression,
 38221 and it participated in the match several times, then the byte offsets in *pmatch*[*i*] shall
 38222 delimit the last such match.

38223 2. If subexpression *i* is not contained within another subexpression, and it did not participate
 38224 in an otherwise successful match, the byte offsets in *pmatch*[*i*] shall be -1. A subexpression
 38225 does not participate in the match when:

38226 '*' or "\{\\" appears immediately after the subexpression in a basic regular
 38227 expression, or '*', '?', or "{ }" appears immediately after the subexpression in an
 38228 extended regular expression, and the subexpression did not match (matched 0 times)

38229 or:

38230 '|' is used in an extended regular expression to select this subexpression or another,
 38231 and the other subexpression matched.

38232 3. If subexpression *i* is contained within another subexpression *j*, and *i* is not contained
 38233 within any other subexpression that is contained within *j*, and a match of subexpression *j*
 38234 is reported in *pmatch*[*j*], then the match or non-match of subexpression *i* reported in
 38235 *pmatch*[*i*] shall be as described in 1. and 2. above, but within the substring reported in
 38236 *pmatch*[*j*] rather than the whole string. The offsets in *pmatch*[*i*] are still relative to the start
 38237 of *string*.

38238 4. If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch*[*j*] are -1,
 38239 then the pointers in *pmatch*[*i*] shall also be -1.

38240 5. If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch*[*i*] shall be
 38241 the byte offset of the character or null terminator immediately following the zero-length
 38242 string.

38243 If, when *regexec()* is called, the locale is different from when the regular expression was
 38244 compiled, the result is undefined.

38245 If REG_NEWLINE is not set in *flags*, then a <newline> in *pattern* or *string* shall be treated as an
38246 ordinary character. If REG_NEWLINE is set, then <newline> shall be treated as an ordinary
38247 character except as follows:

- 38248 1. A <newline> in *string* shall not be matched by a period outside a bracket expression or by
38249 any form of a non-matching list (see the Base Definitions volume of IEEE Std 1003.1-2001,
38250 Chapter 9, Regular Expressions).
- 38251 2. A circumflex ('^') in *pattern*, when used to specify expression anchoring (see the Base
38252 Definitions volume of IEEE Std 1003.1-2001, Section 9.3.8, BRE Expression Anchoring),
38253 shall match the zero-length string immediately after a <newline> in *string*, regardless of
38254 the setting of REG_NOTBOL.
- 38255 3. A dollar sign ('\$') in *pattern*, when used to specify expression anchoring, shall match the
38256 zero-length string immediately before a <newline> in *string*, regardless of the setting of
38257 REG_NOTEOL.

38258 The *regfree()* function frees any memory allocated by *regcomp()* associated with *preg*.

38259 The following constants are defined as error return values:

38260 REG_NOMATCH	<i>regexec()</i> failed to match.
38261 REG_BADPAT	Invalid regular expression.
38262 REG_ECOLLATE	Invalid collating element referenced.
38263 REG_ECTYPE	Invalid character class type referenced.
38264 REG_EESCAPE	Trailing '\' in pattern.
38265 REG_ESUBREG	Number in "\digit" invalid or in error.
38266 REG_EBRACK	"[]" imbalance.
38267 REG_EPAREN	"\(\)" or "()" imbalance.
38268 REG_EBRACE	"\{\}" imbalance.
38269 REG_BADBR	Content of "\{\}" invalid: not a number, number too large, more than 38270 two numbers, first larger than second.
38271 REG_ERANGE	Invalid endpoint in range expression.
38272 REG_ESPACE	Out of memory.
38273 REG_BADRPT	'?' , '*' , or '+' not preceded by valid regular expression.

38274 The *regerror()* function provides a mapping from error codes returned by *regcomp()* and
38275 *regexec()* to unspecified printable strings. It generates a string corresponding to the value of the
38276 *errcode* argument, which the application shall ensure is the last non-zero value returned by
38277 *regcomp()* or *regexec()* with the given value of *preg*. If *errcode* is not such a value, the content of
38278 the generated string is unspecified.

38279 If *preg* is a null pointer, but *errcode* is a value returned by a previous call to *regexec()* or *regcomp()*,
38280 the *regerror()* still generates an error string corresponding to the value of *errcode*, but it might not
38281 be as detailed under some implementations.

38282 If the *errbuf_size* argument is not 0, *regerror()* shall place the generated string into the buffer of
38283 size *errbuf_size* bytes pointed to by *errbuf*. If the string (including the terminating null) cannot fit
38284 in the buffer, *regerror()* shall truncate the string and null-terminate the result.

38285 If *errbuf_size* is 0, *regerror()* shall ignore the *errbuf* argument, and return the size of the buffer
 38286 needed to hold the generated string.

38287 If the *preg* argument to *regexec()* or *regfree()* is not a compiled regular expression returned by
 38288 *regcomp()*, the result is undefined. A *preg* is no longer treated as a compiled regular expression
 38289 after it is given to *regfree()*.

38290 RETURN VALUE

38291 Upon successful completion, the *regcomp()* function shall return 0. Otherwise, it shall return an
 38292 integer value indicating an error as described in <regex.h>, and the content of *preg* is undefined.
 38293 If a code is returned, the interpretation shall be as given in <regex.h>.

38294 If *regcomp()* detects an invalid RE, it may return REG_BADPAT, or it may return one of the error
 38295 codes that more precisely describes the error.

38296 Upon successful completion, the *regexec()* function shall return 0. Otherwise, it shall return
 38297 REG_NOMATCH to indicate no match.

38298 Upon successful completion, the *regerror()* function shall return the number of bytes needed to
 38299 hold the entire generated string, including the null termination. If the return value is greater than
 38300 *errbuf_size*, the string returned in the buffer pointed to by *errbuf* has been truncated.

38301 The *regfree()* function shall not return a value.

38302 ERRORS

38303 No errors are defined.

38304 EXAMPLES

```
38305     #include <regex.h>
38306
38307     /*
38308     * Match string against the extended regular expression in
38309     * pattern, treating errors as no match.
38310     *
38311     * Return 1 for match, 0 for no match.
38312     */
38313
38314     int
38315     match(const char *string, char *pattern)
38316     {
38317         int      status;
38318         regex_t    re;
38319
38320         if (regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB) != 0) {
38321             return(0);          /* Report error. */
38322         }
38323         status = regexec(&re, string, (size_t) 0, NULL, 0);
38324         regfree(&re);
38325         if (status != 0) {
38326             return(0);          /* Report error. */
38327         }
38328         return(1);
38329     }
```

38327 The following demonstrates how the REG_NOTBOL flag could be used with *regexec()* to find all
 38328 substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very
 38329 little error checking is done.)

```
38330     (void) regcomp (&re, pattern, 0);
38331     /* This call to regexec() finds the first match on the line. */
38332     error = regexec (&re, &buffer[0], 1, &pm, 0);
38333     while (error == 0) { /* While matches found. */
38334         /* Substring found between pm.rm_so and pm.rm_eo. */
38335         /* This call to regexec() finds the next match. */
38336         error = regexec (&re, buffer + pm.rm_eo, 1, &pm, REG_NOTBOL);
38337     }
```

38338 APPLICATION USAGE

38339 An application could use:

```
38340     regerror(code, preg, (char *)NULL, (size_t)0)
```

38341 to find out how big a buffer is needed for the generated string, *malloc()* a buffer to hold the
38342 string, and then call *regerror()* again to get the string. Alternatively, it could allocate a fixed,
38343 static buffer that is big enough to hold most strings, and then use *malloc()* to allocate a larger
38344 buffer if it finds that this is too small.

38345 To match a pattern as described in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section
38346 2.13, Pattern Matching Notation, use the *fnmatch()* function.

38347 RATIONALE

38348 The *regexec()* function must fill in all *nmatch* elements of *pmatch*, where *nmatch* and *pmatch* are
38349 supplied by the application, even if some elements of *pmatch* do not correspond to
38350 subexpressions in *pattern*. The application writer should note that there is probably no reason
38351 for using a value of *nmatch* that is larger than *preg->re_nsub+1*.

38352 The REG_NEWLINE flag supports a use of RE matching that is needed in some applications like
38353 text editors. In such applications, the user supplies an RE asking the application to find a line
38354 that matches the given expression. An anchor in such an RE anchors at the beginning or end of
38355 any line. Such an application can pass a sequence of <newline>-separated lines to *regexec()* as a
38356 single long string and specify REG_NEWLINE to *regcomp()* to get the desired behavior. The
38357 application must ensure that there are no explicit <newline>s in *pattern* if it wants to ensure that
38358 any match occurs entirely within a single line.

38359 The REG_NEWLINE flag affects the behavior of *regexec()*, but it is in the *cflags* parameter to
38360 *regcomp()* to allow flexibility of implementation. Some implementations will want to generate
38361 the same compiled RE in *regcomp()* regardless of the setting of REG_NEWLINE and have
38362 *regexec()* handle anchors differently based on the setting of the flag. Other implementations will
38363 generate different compiled REs based on the REG_NEWLINE.

38364 The REG_ICASE flag supports the operations taken by the *grep -i* option and the historical
38365 implementations of *ex* and *vi*. Including this flag will make it easier for application code to be
38366 written that does the same thing as these utilities.

38367 The substrings reported in *pmatch[]* are defined using offsets from the start of the string rather
38368 than pointers. Since this is a new interface, there should be no impact on historical
38369 implementations or applications, and offsets should be just as easy to use as pointers. The
38370 change to offsets was made to facilitate future extensions in which the string to be searched is
38371 presented to *regexec()* in blocks, allowing a string to be searched that is not all in memory at
38372 once.

38373 The type **regoff_t** is used for the elements of *pmatch[]* to ensure that the application can
38374 represent either the largest possible array in memory (important for an application conforming
38375 to the Shell and Utilities volume of IEEE Std 1003.1-2001) or the largest possible file (important
38376 for an application using the extension where a file is searched in chunks).

38377 The standard developers rejected the inclusion of a *regsub()* function that would be used to do
38378 substitutions for a matched RE. While such a routine would be useful to some applications, its
38379 utility would be much more limited than the matching function described here. Both RE parsing
38380 and substitution are possible to implement without support other than that required by the
38381 ISO C standard, but matching is much more complex than substituting. The only difficult part of
38382 substitution, given the information supplied by *regexec()*, is finding the next character in a string
38383 when there can be multi-byte characters. That is a much larger issue, and one that needs a more
38384 general solution.

38385 The *errno* variable has not been used for error returns to avoid filling the *errno* name space for
38386 this feature.

38387 The interface is defined so that the matched substrings *rm_sp* and *rm_ep* are in a separate
38388 *regmatch_t* structure instead of in *regex_t*. This allows a single compiled RE to be used
38389 simultaneously in several contexts; in *main()* and a signal handler, perhaps, or in multiple
38390 threads of lightweight processes. (The *preg* argument to *regexec()* is declared with type **const**, so
38391 the implementation is not permitted to use the structure to store intermediate results.) It also
38392 allows an application to request an arbitrary number of substrings from an RE. The number of
38393 subexpressions in the RE is reported in *re_nsub* in *preg*. With this change to *regexec()*,
38394 consideration was given to dropping the REG_NOSUB flag since the user can now specify this
38395 with a zero *nmatch* argument to *regexec()*. However, keeping REG_NOSUB allows an
38396 implementation to use a different (perhaps more efficient) algorithm if it knows in *regcomp()*
38397 that no subexpressions need be reported. The implementation is only required to fill in *pmatch* if
38398 *nmatch* is not zero and if REG_NOSUB is not specified. Note that the **size_t** type, as defined in
38399 the ISO C standard, is unsigned, so the description of *regexec()* does not need to address
38400 negative values of *nmatch*.

38401 REG_NOTBOL was added to allow an application to do repeated searches for the same pattern
38402 in a line. If the pattern contains a circumflex character that should match the beginning of a line,
38403 then the pattern should only match when matched against the beginning of the line. Without
38404 the REG_NOTBOL flag, the application could rewrite the expression for subsequent matches,
38405 but in the general case this would require parsing the expression. The need for REG_NOTEOL is
38406 not as clear; it was added for symmetry.

38407 The addition of the *regerror()* function addresses the historical need for conforming application
38408 programs to have access to error information more than ‘Function failed to compile/match your
38409 RE for unknown reasons’.

38410 This interface provides for two different methods of dealing with error conditions. The specific
38411 error codes (REG_EBRACE, for example), defined in <*regex.h*>, allow an application to recover
38412 from an error if it is so able. Many applications, especially those that use patterns supplied by a
38413 user, will not try to deal with specific error cases, but will just use *regerror()* to obtain a human-
38414 readable error message to present to the user.

38415 The *regerror()* function uses a scheme similar to *confstr()* to deal with the problem of allocating
38416 memory to hold the generated string. The scheme used by *strror()* in the ISO C standard was
38417 considered unacceptable since it creates difficulties for multi-threaded applications.

38418 The *preg* argument is provided to *regerror()* to allow an implementation to generate a more
38419 descriptive message than would be possible with *errcode* alone. An implementation might, for
38420 example, save the character offset of the offending character of the pattern in a field of *preg*, and
38421 then include that in the generated message string. The implementation may also ignore *preg*.

38422 A REG_FILENAME flag was considered, but omitted. This flag caused *regexec()* to match
38423 patterns as described in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.13,
38424 Pattern Matching Notation instead of REs. This service is now provided by the *fnmatch()*

38425 function.

38426 Notice that there is a difference in philosophy between the ISO POSIX-2:1993 standard and
38427 IEEE Std 1003.1-2001 in how to handle a “bad” regular expression. The ISO POSIX-2:1993
38428 standard says that many bad constructs “produce undefined results”, or that “the interpretation
38429 is undefined”. IEEE Std 1003.1-2001, however, says that the interpretation of such REs is
38430 unspecified. The term “undefined” means that the action by the application is an error, of
38431 similar severity to passing a bad pointer to a function.

38432 The *regcomp()* and *regexec()* functions are required to accept any null-terminated string as the
38433 *pattern* argument. If the meaning of the string is “undefined”, the behavior of the function is
38434 “unspecified”. IEEE Std 1003.1-2001 does not specify how the functions will interpret the
38435 pattern; they might return error codes, or they might do pattern matching in some completely
38436 unexpected way, but they should not do something like abort the process.

38437 FUTURE DIRECTIONS

38438 None.

38439 SEE ALSO

38440 *fnmatch()*, *glob()*, Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.13, Pattern
38441 Matching Notation, Base Definitions volume of IEEE Std 1003.1-2001, Chapter 9, Regular
38442 Expressions, <regex.h>, <sys/types.h>

38443 CHANGE HISTORY

38444 First released in Issue 4. Derived from the ISO POSIX-2 standard.

38445 Issue 5

38446 Moved from POSIX2 C-language Binding to BASE.

38447 Issue 6

38448 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

38449 The following new requirements on POSIX implementations derive from alignment with the
38450 Single UNIX Specification:

- 38451 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
38452 required for conforming implementations of previous POSIX specifications, it was not
38453 required for UNIX applications.

38454 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

38455 The REG_ENOSYS constant is removed.

38456 The **restrict** keyword is added to the *regcomp()*, *regerror()*, and *regexec()* prototypes for
38457 alignment with the ISO/IEC 9899:1999 standard.

38458 NAME

38459 remainder, remainderf, remainderl — remainder function

38460 SYNOPSIS

```
38461 #include <math.h>
38462 double remainder(double x, double y);
38463 float remainderf(float x, float y);
38464 long double remainderl(long double x, long double y);
```

38465 DESCRIPTION

38466 CX 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.

38469 These functions shall return the floating-point remainder $r=x-ny$ when y is non-zero. The value n is the integral value nearest the exact value x/y . When $|n-x/y|=1/2$, the value n is chosen to be even.

38472 The behavior of *remainder()* shall be independent of the rounding mode.

38473 RETURN VALUE

38474 Upon successful completion, these functions shall return the floating-point remainder $r=x-ny$ when y is non-zero.

38476 MX If x or y is NaN, a NaN shall be returned.

38477 If x is infinite or y is 0 and the other is non-NaN, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

38479 ERRORS

38480 These functions shall fail if:

38481 MX Domain Error The x argument is $\pm\text{Inf}$, or the y argument is ± 0 and the other argument is non-NaN.

38483 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.

38487 EXAMPLES

38488 None.

38489 APPLICATION USAGE

38490 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling & MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

38492 RATIONALE

38493 None.

38494 FUTURE DIRECTIONS

38495 None.

38496 SEE ALSO

38497 `abs()`, `div()`, `feclearexcept()`, `fetestexcept()`, `ldiv()`, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, `<math.h>`

38500 CHANGE HISTORY

38501 First released in Issue 4, Version 2.

38502 Issue 5

38503 Moved from X/OPEN UNIX extension to BASE.

38504 Issue 6

38505 The *remainder()* function is no longer marked as an extension.

38506 The *remainderf()* and *remainderl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

38508 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

38510 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

38512 NAME

38513 remove — remove a file

38514 SYNOPSIS

```
38515 #include <stdio.h>
38516 int remove(const char *path);
```

38517 DESCRIPTION

38518 CX The functionality described on this reference page is aligned with the ISO C standard. Any
38519 conflict between the requirements described here and the ISO C standard is unintentional. This
38520 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

38521 The *remove()* function shall cause the file named by the pathname pointed to by *path* to be no
38522 longer accessible by that name. A subsequent attempt to open that file using that name shall fail,
38523 unless it is created anew.

38524 CX If *path* does not name a directory, *remove(path)* shall be equivalent to *unlink(path)*.

38525 If *path* names a directory, *remove(path)* shall be equivalent to *rmdir(path)*.

38526 RETURN VALUE

38527 CX Refer to *rmdir()* or *unlink()*.

38528 ERRORS

38529 CX Refer to *rmdir()* or *unlink()*.

38530 EXAMPLES**38531 Removing Access to a File**

38532 The following example shows how to remove access to a file named */home/cnd/old_mods*.

```
38533 #include <stdio.h>
38534 int status;
38535 ...
38536 status = remove( "/home/cnd/old_mods" );
```

38537 APPLICATION USAGE

38538 None.

38539 RATIONALE

38540 None.

38541 FUTURE DIRECTIONS

38542 None.

38543 SEE ALSO

38544 *rmdir()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

38545 CHANGE HISTORY

38546 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard and the ISO C
38547 standard.

38548 Issue 6

38549 Extensions beyond the ISO C standard are marked.

38550 The following new requirements on POSIX implementations derive from alignment with the
38551 Single UNIX Specification:

38552
38553
38554

- 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()*.

38555 NAME

38556 remque — remove an element from a queue

38557 SYNOPSIS

38558 XSI #include <search.h>

38559 void remque(void *element);

38560

38561 DESCRIPTION

38562 Refer to *insque()*.

38563 **NAME**

38564 remquo, remquof, remquol — remainder functions

38565 **SYNOPSIS**

```
38566     #include <math.h>
38567
38568     double remquo(double x, double y, int *quo);
38569     float remquof(float x, float y, int *quo);
38570     long double remquol(long double x, long double y, int *quo);
```

38570 **DESCRIPTION**

38571 CX 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.

38574 The *remquo()*, *remquof()*, and *remquol()* functions shall compute the same remainder as the *remainder()*, *remainderf()*, and *remainderl()* functions, respectively. In the object pointed to by *quo*, they store a value whose sign is the sign of *x/y* and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of *x/y*, where *n* is an implementation-defined integer greater than or equal to 3.

38579 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.

38583 **RETURN VALUE**

38584 These functions shall return *x REM y*.

38585 MX If *x* or *y* is NaN, a NaN shall be returned.

38586 If *x* is ±Inf or *y* is zero and the other argument is non-NaN, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

38588 **ERRORS**

38589 These functions shall fail if:

38590 MX Domain Error The *x* argument is ±Inf, or the *y* argument is ±0 and the other argument is non-NaN.

38592 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.

38596 **EXAMPLES**

38597 None.

38598 **APPLICATION USAGE**

38599 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

38601 **RATIONALE**

38602 These functions are intended for implementing argument reductions which can exploit a few low-order bits of the quotient. Note that *x* may be so large in magnitude relative to *y* that an exact representation of the quotient is not practical.

38605 FUTURE DIRECTIONS

38606 None.

38607 SEE ALSO

38608 *feclearexcept()*, *fetestexcept()*, *remainder()*, the Base Definitions volume of IEEE Std 1003.1-2001,
38609 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

38610 CHANGE HISTORY

38611 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

38612 NAME

38613 rename — rename a file

38614 SYNOPSIS

38615 #include <stdio.h>
38616 int rename(const char *old, const char *new);

38617 DESCRIPTION

38618 CX The functionality described on this reference page is aligned with the ISO C standard. Any
38619 conflict between the requirements described here and the ISO C standard is unintentional. This
38620 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.38621 The *rename()* function shall change the name of a file. The *old* argument points to the pathname
38622 of the file to be renamed. The *new* argument points to the new pathname of the file.38623 CX If either the *old* or *new* argument names a symbolic link, *rename()* shall operate on the symbolic
38624 link itself, and shall not resolve the last component of the argument. If the *old* argument and the
38625 *new* argument resolve to the same existing file, *rename()* shall return successfully and perform no
38626 other action.38627 If the *old* argument points to the pathname of a file that is not a directory, the *new* argument shall
38628 not point to the pathname of a directory. If the link named by the *new* argument exists, it shall be
38629 removed and *old* renamed to *new*. In this case, a link named *new* shall remain visible to other
38630 processes throughout the renaming operation and refer either to the file referred to by *new* or *old*
38631 before the operation began. Write access permission is required for both the directory containing
38632 *old* and the directory containing *new*.38633 If the *old* argument points to the pathname of a directory, the *new* argument shall not point to the
38634 pathname of a file that is not a directory. If the directory named by the *new* argument exists, it
38635 shall be removed and *old* renamed to *new*. In this case, a link named *new* shall exist throughout
38636 the renaming operation and shall refer either to the directory referred to by *new* or *old* before the
38637 operation began. If *new* names an existing directory, it shall be required to be an empty directory.38638 If the *old* argument points to a pathname of a symbolic link, the symbolic link shall be renamed.
38639 If the *new* argument points to a pathname of a symbolic link, the symbolic link shall be removed.38640 The *new* pathname shall not contain a path prefix that names *old*. Write access permission is
38641 required for the directory containing *old* and the directory containing *new*. If the *old* argument
38642 points to the pathname of a directory, write access permission may be required for the directory
38643 named by *old*, and, if it exists, the directory named by *new*.38644 If the link named by the *new* argument exists and the file's link count becomes 0 when it is
38645 removed and no process has the file open, the space occupied by the file shall be freed and the
38646 file shall no longer be accessible. If one or more processes have the file open when the last link is
38647 removed, the link shall be removed before *rename()* returns, but the removal of the file contents
38648 shall be postponed until all references to the file are closed.38649 Upon successful completion, *rename()* shall mark for update the *st_ctime* and *st_mtime* fields of
38650 the parent directory of each file.38651 If the *rename()* function fails for any reason other than [EIO], any file named by *new* shall be
38652 unaffected.

38653 RETURN VALUE

38654 CX Upon successful completion, *rename()* shall return 0; otherwise, -1 shall be returned, *errno* shall
38655 be set to indicate the error, and neither the file named by *old* nor the file named by *new* shall be
38656 changed or created.

38657 ERRORS

38658 The *rename()* function shall fail if:

38659 CX	[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.
38660 CX	[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.
38661 CX	[EEXIST] or [ENOTEMPTY]	The link named by <i>new</i> is a directory that is not an empty directory.
38662 CX	[EINVAL]	The <i>new</i> directory pathname contains a path prefix that names the <i>old</i> directory.
38663 CX	[EIO]	A physical I/O error has occurred.
38664 CX	[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.
38665 CX	[ELOOP]	A loop exists in symbolic links encountered during resolution of the <i>path</i> argument.
38666 CX	[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}.
38667 CX	[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}.
38668 CX	[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.
38669 CX	[ENOSPC]	The directory that would contain <i>new</i> cannot be extended.
38670 CX	[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.
38671 CX	[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.
38672 CX	[EROFS]	The requested operation requires writing in a directory on a read-only file system.
38673 CX	[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.
38674 CX		The <i>rename()</i> function may fail if:
38675 CX	[EBUSY]	The file named by the <i>old</i> or <i>new</i> arguments is a named STREAM.
38676 CX	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path</i> argument.
38677 CX		
38678 CX		
38679 CX		
38680 CX		
38681 CX		
38682 CX		
38683 CX		
38684 XSI		
38685 XSI		
38686 XSI		
38687 XSI		
38688 XSI		
38689 XSI		
38690 XSI		
38691 XSI		
38692 XSI		
38693 XSI		
38694 XSI		
38695 XSI		
38696 XSI		
38697 XSI		
38698 XSI		

38699 CX [ENAMETOOLONG]	
38700	As a result of encountering a symbolic link in resolution of the <i>path</i> argument,
38701	the length of the substituted pathname string exceeded {PATH_MAX}.
38702 CX [ETXTBSY]	The file to be renamed is a pure procedure (shared text) file that is being
38703	executed.

38704 EXAMPLES

38705 Renaming a File

38706 The following example shows how to rename a file named `/home/cnd/mod1` to
 38707 `/home/cnd/mod2`.

```
38708 #include <stdio.h>
38709 int status;
38710 ...
38711 status = rename( "/home/cnd/mod1" , " /home/cnd/mod2" );
```

38712 APPLICATION USAGE

38713 Some implementations mark for update the *st_ctime* field of renamed files and some do not.
 38714 Applications which make use of the *st_ctime* field may behave differently with respect to
 38715 renamed files unless they are designed to allow for either behavior.

38716 RATIONALE

38717 This `rename()` function is equivalent for regular files to that defined by the ISO C standard. Its
 38718 inclusion here expands that definition to include actions on directories and specifies behavior
 38719 when the *new* parameter names a file that already exists. That specification requires that the
 38720 action of the function be atomic.

38721 One of the reasons for introducing this function was to have a means of renaming directories
 38722 while permitting implementations to prohibit the use of `link()` and `unlink()` with directories,
 38723 thus constraining links to directories to those made by `mkdir()`.

38724 The specification that if *old* and *new* refer to the same file is intended to guarantee that:

```
38725 rename( "x" , "x" );
38726 does not remove the file.
```

38727 Renaming dot or dot-dot is prohibited in order to prevent cyclical file system paths.

38728 See also the descriptions of [ENOTEMPTY] and [ENAMETOOLONG] in `rmdir()` and [EBUSY] in
 38729 `unlink()`. For a discussion of [EXDEV], see `link()`.

38730 FUTURE DIRECTIONS

38731 None.

38732 SEE ALSO

38733 `link()`, `rmdir()`, `symlink()`, `unlink()`, the Base Definitions volume of IEEE Std 1003.1-2001,
 38734 `<stdio.h>`

38735 CHANGE HISTORY

38736 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

38737 Issue 5

38738 The [EBUSY] error is added to the optional part of the ERRORS section.

38739 Issue 6

38740 Extensions beyond the ISO C standard are marked.

38741 The following new requirements on POSIX implementations derive from alignment with the
38742 Single UNIX Specification:

- 38743 • The [EIO] mandatory error condition is added.
- 38744 • The [ELOOP] mandatory error condition is added.
- 38745 • A second [ENAMETOOLONG] is added as an optional error condition.
- 38746 • The [ETXTBSY] optional error condition is added.

38747 The following changes were made to align with the IEEE P1003.1a draft standard:

- 38748 • Details are added regarding the treatment of symbolic links.
- 38749 • The [ELOOP] optional error condition is added.

38750 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

38751 NAME

38752 *rewind* — reset the file position indicator in a stream

38753 SYNOPSIS

```
38754        #include <stdio.h>
38755        void rewind(FILE *stream);
```

38756 DESCRIPTION

38757 CX 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.

38760 The call:

38761 *rewind*(*stream*)

38762 shall be equivalent to:

38763 (*void*) *fseek*(*stream*, 0L, SEEK_SET)

38764 except that *rewind*() shall also clear the error indicator.

38765 CX Since *rewind*() does not return a value, an application wishing to detect errors should clear *errno*, then call *rewind*(), and if *errno* is non-zero, assume an error has occurred.

38767 RETURN VALUE

38768 The *rewind*() function shall not return a value.

38769 ERRORS

38770 CX Refer to *fseek*() with the exception of [EINVAL] which does not apply.

38771 EXAMPLES

38772 None.

38773 APPLICATION USAGE

38774 None.

38775 RATIONALE

38776 None.

38777 FUTURE DIRECTIONS

38778 None.

38779 SEE ALSO

38780 *fseek*(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

38781 CHANGE HISTORY

38782 First released in Issue 1. Derived from Issue 1 of the SVID.

38783 Issue 6

38784 Extensions beyond the ISO C standard are marked.

38785 NAME

38786 **rewinddir** — reset the position of a directory stream to the beginning of a directory

38787 SYNOPSIS

```
38788       #include <dirent.h>
38789       void rewinddir(DIR *dirp);
```

38790 DESCRIPTION

38791 The *rewinddir()* function shall reset the position of the directory stream to which *dirp* refers to
38792 the beginning of the directory. It shall also cause the directory stream to refer to the current state
38793 of the corresponding directory, as a call to *opendir()* would have done. If *dirp* does not refer to a
38794 directory stream, the effect is undefined.

38795 After a call to the *fork()* function, either the parent or child (but not both) may continue
38796 XSI processing the directory stream using *readdir()*, *rewinddir()*, or *seekdir()*. If both the parent and
38797 child processes use these functions, the result is undefined.

38798 RETURN VALUE

38799 The *rewinddir()* function shall not return a value.

38800 ERRORS

38801 No errors are defined.

38802 EXAMPLES

38803 None.

38804 APPLICATION USAGE

38805 The *rewinddir()* function should be used in conjunction with *opendir()*, *readdir()*, and *closedir()* to
38806 examine the contents of the directory. This method is recommended for portability.

38807 RATIONALE

38808 None.

38809 FUTURE DIRECTIONS

38810 None.

38811 SEE ALSO

38812 *closedir()*, *opendir()*, *readdir()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<dirent.h>**
38813 **<sys/types.h>**

38814 CHANGE HISTORY

38815 First released in Issue 2.

38816 Issue 6

38817 In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

38818 The following new requirements on POSIX implementations derive from alignment with the
38819 Single UNIX Specification:

- 38820 • The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was
38821 required for conforming implementations of previous POSIX specifications, it was not
38822 required for UNIX applications.

38823 NAME

38824 rindex — character string operations (**LEGACY**)

38825 SYNOPSIS

38826 XSI #include <strings.h>

38827 char *rindex(const char *s, int c);

38828

38829 DESCRIPTION

38830 The *rindex()* function shall be equivalent to *strrchr()*.

38831 RETURN VALUE

38832 Refer to *strrchr()*.

38833 ERRORS

38834 Refer to *strrchr()*.

38835 EXAMPLES

38836 None.

38837 APPLICATION USAGE

38838 The *strrchr()* function is preferred over this function.

38839 For maximum portability, it is recommended to replace the function call to *rindex()* as follows:

38840 #define rindex(a,b) strrchr((a),(b))

38841 RATIONALE

38842 None.

38843 FUTURE DIRECTIONS

38844 This function may be withdrawn in a future version.

38845 SEE ALSO

38846 *strrchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>

38847 CHANGE HISTORY

38848 First released in Issue 4, Version 2.

38849 Issue 5

38850 Moved from X/OPEN UNIX extension to BASE.

38851 Issue 6

38852 This function is marked LEGACY.

38853 NAME

38854 `rint`, `rintf`, `rintl` — round-to-nearest integral value

38855 SYNOPSIS

```
38856        #include <math.h>
38857        double rint(double x);
38858        float rintf(float x);
38859        long double rintl(long double x);
```

38860 DESCRIPTION

38861 CX 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.

38864 These functions shall return the integral value (represented as a **double**) nearest x in the direction of the current rounding mode. The current rounding mode is implementation-defined.

38866 If the current rounding mode rounds toward negative infinity, then `rint()` shall be equivalent to `floor()`. If the current rounding mode rounds toward positive infinity, then `rint()` shall be equivalent to `ceil()`.

38869 These functions differ from the `nearbyint()`, `nearbyintf()`, and `nearbyintl()` functions only in that they may raise the inexact floating-point exception if the result differs in value from the argument.

38872 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.

38876 RETURN VALUE

38877 Upon successful completion, these functions shall return the integer (represented as a double precision number) nearest x in the direction of the current rounding mode.

38879 MX If x is NaN, a NaN shall be returned.

38880 If x is ± 0 or $\pm \text{Inf}$, x shall be returned.

38881 XSI If the correct value would cause overflow, a range error shall occur and `rint()`, `rintf()`, and `rintl()` shall return the value of the macro `$\pm\text{HUGE_VAL}$` , `$\pm\text{HUGE_VALF}$` , and `$\pm\text{HUGE_VALL}$` (with the same sign as x), respectively.

38884 ERRORS

38885 These functions shall fail if:

38886 XSI Range Error The result would cause an overflow.

38887 If the integer expression (`math_errhandling & MATH_ERRNO`) is non-zero, 38888 then `errno` shall be set to [ERANGE]. If the integer expression 38889 (`math_errhandling & MATH_ERREXCEPT`) is non-zero, then the overflow 3890 floating-point exception shall be raised.

38891 EXAMPLES

38892 None.

38893 APPLICATION USAGE

38894 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
38895 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

38896 RATIONALE

38897 None.

38898 FUTURE DIRECTIONS

38899 None.

38900 SEE ALSO

38901 *abs()*, *ceil()*, *feclearexcept()*, *fetestexcept()*, *floor()*, *isnan()*, *nearbyint()*, the Base Definitions volume
38902 of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
38903 *<math.h>*

38904 CHANGE HISTORY

38905 First released in Issue 4, Version 2.

38906 Issue 5

38907 Moved from X/OPEN UNIX extension to BASE.

38908 Issue 6

38909 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 38910 • The *rintf()* and *rintl()* functions are added.
38911 • The *rint()* function is no longer marked as an extension.
38912 • The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
38913 revised to align with the ISO/IEC 9899:1999 standard.

38914 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
38915 marked.

38916 NAME

38917 rmdir — remove a directory

38918 SYNOPSIS

38919 #include <unistd.h>
38920 int rmdir(const char *path);

38921 DESCRIPTION

38922 The *rmdir()* function shall remove a directory whose name is given by *path*. The directory shall
38923 be removed only if it is an empty directory.

38924 If the directory is the root directory or the current working directory of any process, it is
38925 unspecified whether the function succeeds, or whether it shall fail and set *errno* to [EBUSY].

38926 If *path* names a symbolic link, then *rmdir()* shall fail and set *errno* to [ENOTDIR].

38927 If the *path* argument refers to a path whose final component is either dot or dot-dot, *rmdir()* shall
38928 fail.

38929 If the directory's link count becomes 0 and no process has the directory open, the space occupied
38930 by the directory shall be freed and the directory shall no longer be accessible. If one or more
38931 processes have the directory open when the last link is removed, the dot and dot-dot entries, if
38932 present, shall be removed before *rmdir()* returns and no new entries may be created in the
38933 directory, but the directory shall not be removed until all references to the directory are closed.

38934 If the directory is not an empty directory, *rmdir()* shall fail and set *errno* to [EEXIST] or
38935 [ENOTEMPTY].

38936 Upon successful completion, the *rmdir()* function shall mark for update the *st_ctime* and
38937 *st_mtime* fields of the parent directory.

38938 RETURN VALUE

38939 Upon successful completion, the function *rmdir()* shall return 0. Otherwise, -1 shall be returned,
38940 and *errno* set to indicate the error. If -1 is returned, the named directory shall not be changed.

38941 ERRORS

38942 The *rmdir()* function shall fail if:

38943 [EACCES] Search permission is denied on a component of the path prefix, or write
38944 permission is denied on the parent directory of the directory to be removed.

38945 [EBUSY] The directory to be removed is currently in use by the system or some process
38946 and the implementation considers this to be an error.

38947 [EEXIST] or [ENOTEMPTY]
38948 The *path* argument names a directory that is not an empty directory, or there
38949 are hard links to the directory other than dot or a single entry in dot-dot.

38950 [EINVAL] The *path* argument contains a last component that is dot.

38951 [EIO] A physical I/O error has occurred.

38952 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
38953 argument.

38954 [ENAMETOOLONG]
38955 The length of the *path* argument exceeds {PATH_MAX} or a pathname
38956 component is longer than {NAME_MAX}.

38957 [ENOENT] A component of *path* does not name an existing file, or the *path* argument
38958 names a nonexistent directory or points to an empty string.

38959	[ENOTDIR]	A component of <i>path</i> is not a directory.
38960 XSI	[EPERM] or [EACCES]	The S_ISVTX flag is set on the parent directory of the directory to be removed and the caller is not the owner of the directory to be removed, nor is the caller the owner of the parent directory, nor does the caller have the appropriate privileges.
38965	[EROFS]	The directory entry to be removed resides on a read-only file system.
38966		The <i>rmdir()</i> function may fail if:
38967	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the <i>path</i> argument.
38969	[ENAMETOOLONG]	
38970		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}.
38971		

38972 EXAMPLES

38973 Removing a Directory

38974 The following example shows how to remove a directory named **/home/cnd/mod1**.

```
38975 #include <unistd.h>
38976 int status;
38977 ...
38978 status = rmdir( "/home/cnd/mod1" );
```

38979 APPLICATION USAGE

38980 None.

38981 RATIONALE

38982 The *rmdir()* and *rename()* functions originated in 4.2 BSD, and they used [ENOTEMPTY] for the condition when the directory to be removed does not exist or *new* already exists. When the 1984 /usr/group standard was published, it contained [EEXIST] instead. When these functions were adopted into System V, the 1984 /usr/group standard was used as a reference. Therefore, several existing applications and implementations support/use both forms, and no agreement could be reached on either value. All implementations are required to supply both [EEXIST] and [ENOTEMPTY] in <errno.h> with distinct values, so that applications can use both values in C-language **case** statements.

38990 The meaning of deleting *pathname/dot* is unclear, because the name of the file (directory) in the 38991 parent directory to be removed is not clear, particularly in the presence of multiple links to a 38992 directory.

38993 The POSIX.1-1990 standard was silent with regard to the behavior of *rmdir()* when there are 38994 multiple hard links to the directory being removed. The requirement to set *errno* to [EEXIST] or 38995 [ENOTEMPTY] clarifies the behavior in this case.

38996 If the process' current working directory is being removed, that should be an allowed error.

38997 Virtually all existing implementations detect [ENOTEMPTY] or the case of dot-dot. The text in 38998 Section 2.3 (on page 21) about returning any one of the possible errors permits that behavior to 38999 continue. The [ELOOP] error may be returned if more than {SYMLOOP_MAX} symbolic links 39000 are encountered during resolution of the *path* argument.

39001 FUTURE DIRECTIONS

39002 None.

39003 SEE ALSO

39004 Section 2.3 (on page 21), *mkdir()*, *remove()*, *unlink()*, the Base Definitions volume of
39005 IEEE Std 1003.1-2001, <unistd.h>

39006 CHANGE HISTORY

39007 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

39008 Issue 6

39009 The following new requirements on POSIX implementations derive from alignment with the
39010 Single UNIX Specification:

- 39011 • The DESCRIPTION is updated to indicate the results of naming a symbolic link in *path*.
- 39012 • The [EIO] mandatory error condition is added.
- 39013 • The [ELOOP] mandatory error condition is added.
- 39014 • A second [ENAMETOOLONG] is added as an optional error condition.

39015 The following changes were made to align with the IEEE P1003.1a draft standard:

- 39016 • The [ELOOP] optional error condition is added.

39017 NAME

39018 **round**, **roundf**, **roundl** — round to the nearest integer value in a floating-point format

39019 SYNOPSIS

```
39020       #include <math.h>
39021       double round(double x);
39022       float roundf(float x);
39023       long double roundl(long double x);
```

39024 DESCRIPTION

39025 CX The functionality described on this reference page is aligned with the ISO C standard. Any
39026 conflict between the requirements described here and the ISO C standard is unintentional. This
39027 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

39028 These functions shall round their argument to the nearest integer value in floating-point format,
39029 rounding halfway cases away from zero, regardless of the current rounding direction.

39030 An application wishing to check for error situations should set *errno* to zero and call
39031 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
39032 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
39033 zero, an error has occurred.

39034 RETURN VALUE

39035 Upon successful completion, these functions shall return the rounded integer value.

39036 MX If *x* is NaN, a NaN shall be returned.

39037 If *x* is ±0 or ±Inf, *x* shall be returned.

39038 XSI If the correct value would cause overflow, a range error shall occur and *round()*, *roundf()*, and
39039 *roundl()* shall return the value of the macro ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL
39040 (with the same sign as *x*), respectively.

39041 ERRORS

39042 These functions may fail if:

39043 XSI Range Error The result overflows.

39044 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
39045 then *errno* shall be set to [ERANGE]. If the integer expression
39046 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
39047 floating-point exception shall be raised.

39048 EXAMPLES

39049 None.

39050 APPLICATION USAGE

39051 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
39052 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

39053 RATIONALE

39054 None.

39055 FUTURE DIRECTIONS

39056 None.

39057 SEE ALSO

39058 *feclearexcept()*, *fetestexcept()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18,
39059 Treatment of Error Conditions for Mathematical Functions, <math.h>

39060 CHANGE HISTORY

39061 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

39062 NAME

39063 scalb — load exponent of a radix-independent floating-point number

39064 SYNOPSIS

39065 OB XSI #include <math.h>

```
39066     double scalb(double x, double n);
```

39067

39068 DESCRIPTION

39069 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>.

39072 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.

39076 RETURN VALUE

39077 Upon successful completion, the *scalb()* function shall return $x \cdot r^n$.

39078 If x or n is NaN, a NaN shall be returned.

39079 If n is zero, x shall be returned.

39080 If x is ±Inf and n is not –Inf, x shall be returned.

39081 If x is ±0 and n is not +Inf, x shall be returned.

39082 If x is ±0 and n is +Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

39084 If x is ±Inf and n is –Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

39086 If the result would cause an overflow, a range error shall occur and ±HUGE_VAL (according to the sign of x) shall be returned.

39088 If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

39090 If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 shall be returned.

39092 ERRORS

39093 The *scalb()* function shall fail if:

39094 Domain Error If x is zero and n is +Inf, or x is Inf and n is –Inf.

39095 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.

39099 Range Error The result would overflow.

39100 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.

39104 The *scalb()* function may fail if:

39105 Range Error The result underflows.

39106 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
39107 then *errno* shall be set to [ERANGE]. If the integer expression
39108 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
39109 floating-point exception shall be raised.

39110 EXAMPLES

39111 None.

39112 APPLICATION USAGE

39113 Applications should use either *scalbln()*, *scalblnf()*, or *scalblnl()* in preference to this function.

39114 IEEE Std 1003.1-2001 only defines the behavior for the *scalb()* function when the *n* argument is
39115 an integer, a NaN, or Inf. The behavior of other values for the *n* argument is unspecified.

39116 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* &
39117 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

39118 RATIONALE

39119 None.

39120 FUTURE DIRECTIONS

39121 None.

39122 SEE ALSO

39123 *feclearexcept()*, *fetestexcept()*, *ilogb()*, *ldexp()*, *logb()*, *scalbln()*, the Base Definitions volume of
39124 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
39125 <float.h>, <math.h>

39126 CHANGE HISTORY

39127 First released in Issue 4, Version 2.

39128 Issue 5

39129 Moved from X/OPEN UNIX extension to BASE.

39130 The DESCRIPTION is updated to indicate how an application should check for an error. This
39131 text was previously published in the APPLICATION USAGE section.

39132 Issue 6

39133 This function is marked obsolescent.

39134 Although this function is not part of the ISO/IEC 9899: 1999 standard, the RETURN VALUE and
39135 ERRORS sections are updated to align with the error handling in the ISO/IEC 9899: 1999
39136 standard.

39137 NAME

39138 scalbln, scalblnf, scalblnl, scalbn, scalbnf, scalbnl — compute exponent using FLT_RADIX

39139 SYNOPSIS

```
39140     #include <math.h>
39141
39142     double scalbln(double x, long n);
39143     float scalblnf(float x, long n);
39144     long double scalblnl(long double x, long n);
39145     double scalbn(double x, int n);
39146     float scalbnf(float x, int n);
39147     long double scalbnl(long double x, int n);
```

39147 DESCRIPTION

39148 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 39149 conflict between the requirements described here and the ISO C standard is unintentional. This
 39150 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

39151 These functions shall compute $x * \text{FLT_RADIX}^n$ efficiently, not normally by computing
 39152 FLT_RADIX^n explicitly.

39153 An application wishing to check for error situations should set *errno* to zero and call
 39154 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
 39155 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
 39156 zero, an error has occurred.

39157 RETURN VALUE

39158 Upon successful completion, these functions shall return $x * \text{FLT_RADIX}^n$.

39159 If the result would cause overflow, a range error shall occur and these functions shall return
 39160 $\pm\text{HUGE_VAL}$, $\pm\text{HUGE_VALF}$, and $\pm\text{HUGE_VALL}$ (according to the sign of *x*) as appropriate for
 39161 the return type of the function.

39162 If the correct value would cause underflow, and is not representable, a range error may occur,
 39163 MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

39164 MX If *x* is NaN, a NaN shall be returned.

39165 If *x* is ± 0 or $\pm\text{Inf}$, *x* shall be returned.

39166 If *n* is 0, *x* shall be returned.

39167 If the correct value would cause underflow, and is representable, a range error may occur and
 39168 the correct value shall be returned.

39169 ERRORS

39170 These functions shall fail if:

39171 Range Error The result overflows.

39172 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 39173 then *errno* shall be set to [ERANGE]. If the integer expression
 39174 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 39175 floating-point exception shall be raised.

39176 These functions may fail if:

39177 Range Error The result underflows.

39178 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 39179 then *errno* shall be set to [ERANGE]. If the integer expression

39180 (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
39181 floating-point exception shall be raised.

39182 EXAMPLES

39183 None.

39184 APPLICATION USAGE

39185 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling & MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

39187 RATIONALE

These functions are named so as to avoid conflicting with the historical definition of the `scalb()` function from the Single UNIX Specification. The difference is that the `scalb()` function has a second argument of **double** instead of **int**. The `scalb()` function is not part of the ISO C standard. The three functions whose second type is **long** are provided because the factor required to scale from the smallest positive floating-point value to the largest finite one, on many implementations, is too large to represent in the minimum-width **int** format.

39194 FUTURE DIRECTIONS

39195 None.

39196 SEE ALSO

39197 *feclearexcept(), fetestexcept(), scalb()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section
39198 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

39199 CHANGE HISTORY

39200 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

39201 **NAME**
39202 `scanf` — convert formatted input

39203 **SYNOPSIS**
39204 `#include <stdio.h>`
39205 `int scanf(const char *restrict format, ...);`

39206 **DESCRIPTION**
39207 Refer to *fscanf()*.

39208 NAME

39209 sched_get_priority_max, sched_get_priority_min — get priority limits (REALTIME)

39210 SYNOPSIS

39211 PS|TPS #include <sched.h>

1

```
39212        int sched_get_priority_max(int policy);
```

```
39213        int sched_get_priority_min(int policy);
```

```
39214
```

39215 DESCRIPTION

39216 The *sched_get_priority_max()* and *sched_get_priority_min()* functions shall return the appropriate
39217 maximum or minimum, respectively, for the scheduling policy specified by *policy*.

39218 The value of *policy* shall be one of the scheduling policy values defined in <sched.h>.

39219 RETURN VALUE

39220 If successful, the *sched_get_priority_max()* and *sched_get_priority_min()* functions shall return the
39221 appropriate maximum or minimum values, respectively. If unsuccessful, they shall return a
39222 value of -1 and set *errno* to indicate the error.

39223 ERRORS

39224 The *sched_get_priority_max()* and *sched_get_priority_min()* functions shall fail if:

39225 [EINVAL] The value of the *policy* parameter does not represent a defined scheduling
39226 policy.

39227 EXAMPLES

39228 None.

39229 APPLICATION USAGE

39230 None.

39231 RATIONALE

39232 None.

39233 FUTURE DIRECTIONS

39234 None.

39235 SEE ALSO

39236 *sched_getparam()*, *sched_setparam()*, *sched_getscheduler()*, *sched_rr_get_interval()*,
39237 *sched_setscheduler()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sched.h>

39238 CHANGE HISTORY

39239 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39240 Issue 6

39241 These functions are marked as part of the Process Scheduling option.

39242 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39243 implementation does not support the Process Scheduling option.

39244 The [ESRCH] error condition has been removed since these functions do not take a *pid*
39245 argument.

39246 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/52 is applied, changing the PS margin 1
39247 code in the SYNOPSIS to PS|TPS. 1

39248 NAME

39249 *sched_getparam* — get scheduling parameters (**REALTIME**)

39250 SYNOPSIS

39251 PS

```
#include <sched.h>
```


39252

```
int sched_getparam(pid_t pid, struct sched_param *param);
```


39253

39254 DESCRIPTION

39255 The *sched_getparam()* function shall return the scheduling parameters of a process specified by
39256 *pid* in the **sched_param** structure pointed to by *param*.

39257 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
39258 parameters for the process whose process ID is equal to *pid* shall be returned.

39259 If *pid* is zero, the scheduling parameters for the calling process shall be returned. The behavior of
39260 the *sched_getparam()* function is unspecified if the value of *pid* is negative.

39261 RETURN VALUE

39262 Upon successful completion, the *sched_getparam()* function shall return zero. If the call to
39263 *sched_getparam()* is unsuccessful, the function shall return a value of -1 and set *errno* to indicate
39264 the error.

39265 ERRORS

39266 The *sched_getparam()* function shall fail if:

39267 [EPERM] The requesting process does not have permission to obtain the scheduling
39268 parameters of the specified process.

39269 [ESRCH] No process can be found corresponding to that specified by *pid*.

39270 EXAMPLES

39271 None.

39272 APPLICATION USAGE

39273 None.

39274 RATIONALE

39275 None.

39276 FUTURE DIRECTIONS

39277 None.

39278 SEE ALSO

39279 *sched_getscheduler()*, *sched_setparam()*, *sched_setscheduler()*, the Base Definitions volume of
39280 IEEE Std 1003.1-2001, <**sched.h**>

39281 CHANGE HISTORY

39282 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39283 Issue 6

39284 The *sched_getparam()* function is marked as part of the Process Scheduling option.

39285 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39286 implementation does not support the Process Scheduling option.

39287 **NAME**

39288 sched_getscheduler — get scheduling policy (**REALTIME**)

39289 **SYNOPSIS**

39290 PS #include <sched.h>

39291 int sched_getscheduler(pid_t pid);

39292

39293 **DESCRIPTION**

39294 The *sched_getscheduler()* function shall return the scheduling policy of the process specified by *pid*. If the value of *pid* is negative, the behavior of the *sched_getscheduler()* function is unspecified.

39297 The values that can be returned by *sched_getscheduler()* are defined in the <**sched.h**> header.

39298 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
39299 policy shall be returned for the process whose process ID is equal to *pid*.

39300 If *pid* is zero, the scheduling policy shall be returned for the calling process.

39301 **RETURN VALUE**

39302 Upon successful completion, the *sched_getscheduler()* function shall return the scheduling policy
39303 of the specified process. If unsuccessful, the function shall return -1 and set *errno* to indicate the
39304 error.

39305 **ERRORS**

39306 The *sched_getscheduler()* function shall fail if:

39307 [EPERM] The requesting process does not have permission to determine the scheduling
39308 policy of the specified process.

39309 [ESRCH] No process can be found corresponding to that specified by *pid*.

39310 **EXAMPLES**

39311 None.

39312 **APPLICATION USAGE**

39313 None.

39314 **RATIONALE**

39315 None.

39316 **FUTURE DIRECTIONS**

39317 None.

39318 **SEE ALSO**

39319 *sched_getparam()*, *sched_setparam()*, *sched_setscheduler()*, the Base Definitions volume of
39320 IEEE Std 1003.1-2001, <**sched.h**>

39321 **CHANGE HISTORY**

39322 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39323 **Issue 6**

39324 The *sched_getscheduler()* function is marked as part of the Process Scheduling option.

39325 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39326 implementation does not support the Process Scheduling option.

39327 **NAME**

39328 sched_rr_get_interval — get execution time limits (**REALTIME**)

39329 **SYNOPSIS**

39330 PS|TPS #include <sched.h>

39331 int sched_rr_get_interval(pid_t pid, struct timespec *interval);

39332

39333 **DESCRIPTION**

39334 The *sched_rr_get_interval()* function shall update the **timespec** structure referenced by the *interval* argument to contain the current execution time limit (that is, time quantum) for the process specified by *pid*. If *pid* is zero, the current execution time limit for the calling process shall be returned.

39338 **RETURN VALUE**

39339 If successful, the *sched_rr_get_interval()* function shall return zero. Otherwise, it shall return a value of -1 and set *errno* to indicate the error.

39341 **ERRORS**

39342 The *sched_rr_get_interval()* function shall fail if:

39343 [ESRCH] No process can be found corresponding to that specified by *pid*.

39344 **EXAMPLES**

39345 None.

39346 **APPLICATION USAGE**

39347 None.

39348 **RATIONALE**

39349 None.

39350 **FUTURE DIRECTIONS**

39351 None.

39352 **SEE ALSO**

39353 *sched_getparam()*, *sched_get_priority_max()*, *sched_getscheduler()*, *sched_setparam()*,
39354 *sched_setscheduler()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sched.h>

39355 **CHANGE HISTORY**

39356 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39357 **Issue 6**

39358 The *sched_rr_get_interval()* function is marked as part of the Process Scheduling option.

39359 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39360 implementation does not support the Process Scheduling option.

39361 IEEE Std 1003.1-2001/Cor 1-2002, XSH/TC1/D6/53 is applied, changing the PS margin code in 1
39362 the SYNOPSIS to PS|TPS. 1

39363 NAME

39364 sched_setparam — set scheduling parameters (REALTIME)

39365 SYNOPSIS

39366 PS #include <sched.h>

39367 int sched_setparam(pid_t pid, const struct sched_param *param);

39368

39369 DESCRIPTION

39370 The *sched_setparam()* function shall set the scheduling parameters of the process specified by *pid*
 39371 to the values specified by the **sched_param** structure pointed to by *param*. The value of the
 39372 *sched_priority* member in the **sched_param** structure shall be any integer within the inclusive
 39373 priority range for the current scheduling policy of the process specified by *pid*. Higher
 39374 numerical values for the priority represent higher priorities. If the value of *pid* is negative, the
 39375 behavior of the *sched_setparam()* function is unspecified.

39376 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
 39377 parameters shall be set for the process whose process ID is equal to *pid*.

39378 If *pid* is zero, the scheduling parameters shall be set for the calling process.

39379 The conditions under which one process has permission to change the scheduling parameters of
 39380 another process are implementation-defined.

39381 Implementations may require the requesting process to have the appropriate privilege to set its
 39382 own scheduling parameters or those of another process.

39383 The target process, whether it is running or not running, shall be moved to the tail of the thread
 39384 list for its priority.

39385 If the priority of the process specified by the *pid* argument is set higher than that of the lowest
 39386 priority running process and if the specified process is ready to run, the process specified by the
 39387 *pid* argument shall preempt a lowest priority running process. Similarly, if the process calling
 39388 *sched_setparam()* sets its own priority lower than that of one or more other non-empty process
 39389 lists, then the process that is the head of the highest priority list shall also preempt the calling
 39390 process. Thus, in either case, the originating process might not receive notification of the
 39391 completion of the requested priority change until the higher priority process has executed.

39392 SS If the scheduling policy of the target process is SCHED_SPORADIC, the value specified by the
 39393 *sched_ss_low_priority* member of the *param* argument shall be any integer within the inclusive
 39394 priority range for the sporadic server policy. The *sched_ss_repl_period* and *sched_ss_init_budget*
 39395 members of the *param* argument shall represent the time parameters to be used by the sporadic
 39396 server scheduling policy for the target process. The *sched_ss_max_repl* member of the *param*
 39397 argument shall represent the maximum number of replenishments that are allowed to be
 39398 pending simultaneously for the process scheduled under this scheduling policy.

39399 The specified *sched_ss_repl_period* shall be greater than or equal to the specified
 39400 *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

39401 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
 39402 function to succeed; if not, the function shall fail.

39403 If the scheduling policy of the target process is either SCHED_FIFO or SCHED_RR, the
 39404 *sched_ss_low_priority*, *sched_ss_repl_period*, and *sched_ss_init_budget* members of the *param*
 39405 argument shall have no effect on the scheduling behavior. If the scheduling policy of this process
 39406 is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the effects of these members are
 39407 implementation-defined; this case includes the SCHED_OTHER policy.

39408 If the current scheduling policy for the process specified by *pid* is not SCHED_FIFO,
39409 SS, SCHED_RR, or SCHED_SPORADIC, the result is implementation-defined; this case includes the
39410 SCHED_OTHER policy.

39411 The effect of this function on individual threads is dependent on the scheduling contention
39412 scope of the threads:

- 39413 • For threads with system scheduling contention scope, these functions shall have no effect on
39414 their scheduling.
- 39415 • For threads with process scheduling contention scope, the threads' scheduling parameters
39416 shall not be affected. However, the scheduling of these threads with respect to threads in
39417 other processes may be dependent on the scheduling parameters of their process, which are
39418 governed using these functions.

39419 If an implementation supports a two-level scheduling model in which library threads are
39420 multiplexed on top of several kernel-scheduled entities, then the underlying kernel-scheduled
39421 entities for the system contention scope threads shall not be affected by these functions.

39422 The underlying kernel-scheduled entities for the process contention scope threads shall have
39423 their scheduling parameters changed to the value specified in *param*. Kernel-scheduled entities
39424 for use by process contention scope threads that are created after this call completes shall inherit
39425 their scheduling policy and associated scheduling parameters from the process.

39426 This function is not atomic with respect to other threads in the process. Threads may continue to
39427 execute while this function call is in the process of changing the scheduling policy for the
39428 underlying kernel-scheduled entities used by the process contention scope threads.

39429 RETURN VALUE

39430 If successful, the *sched_setparam()* function shall return zero.

39431 If the call to *sched_setparam()* is unsuccessful, the priority shall remain unchanged, and the
39432 function shall return a value of -1 and set *errno* to indicate the error.

39433 ERRORS

39434 The *sched_setparam()* function shall fail if:

- 39435 [EINVAL] One or more of the requested scheduling parameters is outside the range
39436 defined for the scheduling policy of the specified *pid*.
- 39437 [EPERM] The requesting process does not have permission to set the scheduling
39438 parameters for the specified process, or does not have the appropriate
39439 privilege to invoke *sched_setparam()*.
- 39440 [ESRCH] No process can be found corresponding to that specified by *pid*.

39441 EXAMPLES

39442 None.

39443 APPLICATION USAGE

39444 None.

39445 RATIONALE

39446 None.

39447 FUTURE DIRECTIONS

39448 None.

39449 SEE ALSO

39450 *sched_getparam()*, *sched_getscheduler()*, *sched_setscheduler()*, the Base Definitions volume of
39451 IEEE Std 1003.1-2001, <**sched.h**>

39452 CHANGE HISTORY

39453 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39454 Issue 6

39455 The *sched_setparam()* function is marked as part of the Process Scheduling option.

39456 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39457 implementation does not support the Process Scheduling option.

39458 The following new requirements on POSIX implementations derive from alignment with the
39459 Single UNIX Specification:

- 39460 • In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is
39461 added.
- 39462 • Sections describing two-level scheduling and atomicity of the function are added.

39463 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

39464 IEEE PASC Interpretation 1003.1 #100 is applied.

39465 NAME

39466 sched_setscheduler — set scheduling policy and parameters (REALTIME)

39467 SYNOPSIS

39468 PS #include <sched.h>

39469 int sched_setscheduler(pid_t pid, int policy,
39470 const struct sched_param *param);

39471

39472 DESCRIPTION

39473 The *sched_setscheduler()* function shall set the scheduling policy and scheduling parameters of
 39474 the process specified by *pid* to *policy* and the parameters specified in the **sched_param** structure
 39475 pointed to by *param*, respectively. The value of the *sched_priority* member in the **sched_param**
 39476 structure shall be any integer within the inclusive priority range for the scheduling policy
 39477 specified by *policy*. If the value of *pid* is negative, the behavior of the *sched_setscheduler()*
 39478 function is unspecified.

39479 The possible values for the *policy* parameter are defined in the <**sched.h**> header.

39480 If a process specified by *pid* exists, and if the calling process has permission, the scheduling
 39481 policy and scheduling parameters shall be set for the process whose process ID is equal to *pid*.

39482 If *pid* is zero, the scheduling policy and scheduling parameters shall be set for the calling
 39483 process.

39484 The conditions under which one process has the appropriate privilege to change the scheduling
 39485 parameters of another process are implementation-defined.

39486 Implementations may require that the requesting process have permission to set its own
 39487 scheduling parameters or those of another process. Additionally, implementation-defined
 39488 restrictions may apply as to the appropriate privileges required to set a process' own scheduling
 39489 policy, or another process' scheduling policy, to a particular value.

39490 The *sched_setscheduler()* function shall be considered successful if it succeeds in setting the
 39491 scheduling policy and scheduling parameters of the process specified by *pid* to the values
 39492 specified by *policy* and the structure pointed to by *param*, respectively.

39493 SS If the scheduling policy specified by *policy* is **SCHED_SPORADIC**, the value specified by the
 39494 *sched_ss_low_priority* member of the *param* argument shall be any integer within the inclusive
 39495 priority range for the sporadic server policy. The *sched_ss_repl_period* and *sched_ss_init_budget*
 39496 members of the *param* argument shall represent the time parameters used by the sporadic server
 39497 scheduling policy for the target process. The *sched_ss_max_repl* member of the *param* argument
 39498 shall represent the maximum number of replenishments that are allowed to be pending
 39499 simultaneously for the process scheduled under this scheduling policy.

39500 The specified *sched_ss_repl_period* shall be greater than or equal to the specified
 39501 *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

39502 The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the
 39503 function to succeed; if not, the function shall fail.

39504 If the scheduling policy specified by *policy* is either **SCHED_FIFO** or **SCHED_RR**, the
 39505 *sched_ss_low_priority*, *sched_ss_repl_period*, and *sched_ss_init_budget* members of the *param*
 39506 argument shall have no effect on the scheduling behavior.

39507 The effect of this function on individual threads is dependent on the scheduling contention
 39508 scope of the threads:

- 39509 • For threads with system scheduling contention scope, these functions shall have no effect on
39510 their scheduling.
39511 • For threads with process scheduling contention scope, the threads' scheduling policy and
39512 associated parameters shall not be affected. However, the scheduling of these threads with
39513 respect to threads in other processes may be dependent on the scheduling parameters of their
39514 process, which are governed using these functions.

39515 If an implementation supports a two-level scheduling model in which library threads are
39516 multiplexed on top of several kernel-scheduled entities, then the underlying kernel-scheduled
39517 entities for the system contention scope threads shall not be affected by these functions.

39518 The underlying kernel-scheduled entities for the process contention scope threads shall have
39519 their scheduling policy and associated scheduling parameters changed to the values specified in
39520 and *param*, respectively. Kernel-scheduled entities for use by process contention scope
39521 threads that are created after this call completes shall inherit their scheduling policy and
39522 associated scheduling parameters from the process.

39523 This function is not atomic with respect to other threads in the process. Threads may continue to
39524 execute while this function call is in the process of changing the scheduling policy and
39525 associated scheduling parameters for the underlying kernel-scheduled entities used by the
39526 process contention scope threads.

39527 RETURN VALUE

39528 Upon successful completion, the function shall return the former scheduling policy of the
39529 specified process. If the *sched_setscheduler()* function fails to complete successfully, the policy
39530 and scheduling parameters shall remain unchanged, and the function shall return a value of -1
39531 and set *errno* to indicate the error.

39532 ERRORS

39533 The *sched_setscheduler()* function shall fail if:

39534 [EINVAL] The value of the *policy* parameter is invalid, or one or more of the parameters
39535 contained in *param* is outside the valid range for the specified scheduling
39536 policy.

39537 [EPERM] The requesting process does not have permission to set either or both of the
39538 scheduling parameters or the scheduling policy of the specified process.

39539 [ESRCH] No process can be found corresponding to that specified by *pid*.

39540 EXAMPLES

39541 None.

39542 APPLICATION USAGE

39543 None.

39544 RATIONALE

39545 None.

39546 FUTURE DIRECTIONS

39547 None.

39548 SEE ALSO

39549 *sched_getparam()*, *sched_getscheduler()*, *sched_setparam()*, the Base Definitions volume of
39550 IEEE Std 1003.1-2001, <sched.h>

39551 CHANGE HISTORY

39552 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39553 Issue 6

39554 The *sched_setscheduler()* function is marked as part of the Process Scheduling option.

39555 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39556 implementation does not support the Process Scheduling option.

39557 The following new requirements on POSIX implementations derive from alignment with the
39558 Single UNIX Specification:

- 39559 • In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is
39560 added.

- 39561 • Sections describing two-level scheduling and atomicity of the function are added.

39562 The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

39563 NAME

39564 *sched_yield* — yield the processor

39565 SYNOPSIS

39566 PS|THR #include <sched.h>

39567 int sched_yield(void);

39568

39569 DESCRIPTION

39570 The *sched_yield()* function shall force the running thread to relinquish the processor until it again becomes the head of its thread list. It takes no arguments.

39572 RETURN VALUE

39573 The *sched_yield()* function shall return 0 if it completes successfully; otherwise, it shall return a value of -1 and set *errno* to indicate the error.

39575 ERRORS

39576 No errors are defined.

39577 EXAMPLES

39578 None.

39579 APPLICATION USAGE

39580 None.

39581 RATIONALE

39582 None.

39583 FUTURE DIRECTIONS

39584 None.

39585 SEE ALSO

39586 The Base Definitions volume of IEEE Std 1003.1-2001, <**sched.h**>

39587 CHANGE HISTORY

39588 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
39589 POSIX Threads Extension.

39590 Issue 6

39591 The *sched_yield()* function is now marked as part of the Process Scheduling and Threads options.

39592 NAME

39593 seed48 — seed a uniformly distributed pseudo-random non-negative long integer generator

39594 SYNOPSIS

39595 XSI #include <stdlib.h>

39596 unsigned short *seed48(unsigned short seed16v[3]);

39597

39598 DESCRIPTION

39599 Refer to *drand48()*.

39600 NAME

39601 seekdir — set the position of a directory stream

39602 SYNOPSIS

39603 XSI #include <dirent.h>

39604 void seekdir(DIR *dirp, long loc);

39605

39606 DESCRIPTION

39607 The *seekdir()* function shall set the position of the next *readdir()* operation on the directory stream specified by *dirp* to the position specified by *loc*. The value of *loc* should have been returned from an earlier call to *telldir()*. The new position reverts to the one associated with the directory stream when *telldir()* was performed.

39611 If the value of *loc* was not obtained from an earlier call to *telldir()*, or if a call to *rewinddir()* occurred between the call to *telldir()* and the call to *seekdir()*, the results of subsequent calls to *readdir()* are unspecified.

39614 RETURN VALUE

39615 The *seekdir()* function shall not return a value.

39616 ERRORS

39617 No errors are defined.

39618 EXAMPLES

39619 None.

39620 APPLICATION USAGE

39621 None.

39622 RATIONALE

39623 The original standard developers perceived that there were restrictions on the use of the *seekdir()* and *telldir()* functions related to implementation details, and for that reason these functions need not be supported on all POSIX-conforming systems. They are required on implementations supporting the XSI extension.

39627 One of the perceived problems of implementation is that returning to a given point in a directory is quite difficult to describe formally, in spite of its intuitive appeal, when systems that use B-trees, hashing functions, or other similar mechanisms to order their directories are considered. The definition of *seekdir()* and *telldir()* does not specify whether, when using these interfaces, a given directory entry will be seen at all, or more than once.

39632 On systems not supporting these functions, their capability can sometimes be accomplished by saving a filename found by *readdir()* and later using *rewinddir()* and a loop on *readdir()* to relocate the position from which the filename was saved.

39635 FUTURE DIRECTIONS

39636 None.

39637 SEE ALSO

39638 *opendir()*, *readdir()*, *telldir()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<dirent.h>**,
39639 **<stdio.h>**, **<sys/types.h>**

39640 CHANGE HISTORY

39641 First released in Issue 2.

39642 **Issue 6**

39643

In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.

39644 **NAME**39645 **select** — synchronous I/O multiplexing39646 **SYNOPSIS**39647

```
#include <sys/select.h>
```

2

39648

```
int select(int nfds, fd_set *restrict readfds,
```

39649 `fd_set *restrict writefds, fd_set *restrict errorfds,`39650 `struct timeval *restrict timeout);`

39651

39652 **DESCRIPTION**39653 Refer to *pselect()*.

39654 NAME

39655 *sem_close* — close a named semaphore (**REALTIME**)

39656 SYNOPSIS

39657 SEM *#include <semaphore.h>*

39658 *int sem_close(sem_t *sem);*

39659

39660 DESCRIPTION

39661 The *sem_close()* function shall indicate that the calling process is finished using the named
39662 semaphore indicated by *sem*. The effects of calling *sem_close()* for an unnamed semaphore (one
39663 created by *sem_init()*) are undefined. The *sem_close()* function shall deallocate (that is, make
39664 available for reuse by a subsequent *sem_open()* by this process) any system resources allocated
39665 by the system for use by this process for this semaphore. The effect of subsequent use of the
39666 semaphore indicated by *sem* by this process is undefined. If the semaphore has not been
39667 removed with a successful call to *sem_unlink()*, then *sem_close()* has no effect on the state of the
39668 semaphore. If the *sem_unlink()* function has been successfully invoked for *name* after the most
39669 recent call to *sem_open()* with O_CREAT for this semaphore, then when all processes that have
39670 opened the semaphore close it, the semaphore is no longer accessible.

39671 RETURN VALUE

39672 Upon successful completion, a value of zero shall be returned. Otherwise, a value of -1 shall be
39673 returned and *errno* set to indicate the error.

39674 ERRORS

39675 The *sem_close()* function may fail if:

2

39676 [EINVAL] The *sem* argument is not a valid semaphore descriptor.

39677 EXAMPLES

39678 None.

39679 APPLICATION USAGE

39680 The *sem_close()* function is part of the Semaphores option and need not be available on all
39681 implementations.

39682 RATIONALE

39683 None.

39684 FUTURE DIRECTIONS

39685 None.

39686 SEE ALSO

39687 *semctl()*, *semget()*, *semop()*, *sem_init()*, *sem_open()*, *sem_unlink()*, the Base Definitions volume of
39688 IEEE Std 1003.1-2001, <semaphore.h>

39689 CHANGE HISTORY

39690 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39691 Issue 6

39692 The *sem_close()* function is marked as part of the Semaphores option.

39693 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39694 implementation does not support the Semaphores option.

39695 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/113 is applied, updating the ERRORS 2
39696 section so that the [EINVAL] error becomes optional. 2

39697 NAME

39698 sem_destroy — destroy an unnamed semaphore (REALTIME)

39699 SYNOPSIS

39700 SEM #include <semaphore.h>

39701 int sem_destroy(sem_t *sem);

39702

39703 DESCRIPTION

39704 The *sem_destroy()* function shall destroy the unnamed semaphore indicated by *sem*. Only a
39705 semaphore that was created using *sem_init()* may be destroyed using *sem_destroy()*; the effect of
39706 calling *sem_destroy()* with a named semaphore is undefined. The effect of subsequent use of the
39707 semaphore *sem* is undefined until *sem* is reinitialized by another call to *sem_init()*.

39708 It is safe to destroy an initialized semaphore upon which no threads are currently blocked. The
39709 effect of destroying a semaphore upon which other threads are currently blocked is undefined.

39710 RETURN VALUE

39711 Upon successful completion, a value of zero shall be returned. Otherwise, a value of -1 shall be
39712 returned and *errno* set to indicate the error.

39713 ERRORS

39714 The *sem_destroy()* function may fail if:

2

39715 [EINVAL] The *sem* argument is not a valid semaphore.

2

39716 [EBUSY] There are currently processes blocked on the semaphore.

39717 EXAMPLES

39718 None.

39719 APPLICATION USAGE

39720 The *sem_destroy()* function is part of the Semaphores option and need not be available on all
39721 implementations.

39722 RATIONALE

39723 None.

39724 FUTURE DIRECTIONS

39725 None.

39726 SEE ALSO

39727 *semctl()*, *semget()*, *semop()*, *sem_init()*, *sem_open()*, the Base Definitions volume of
39728 IEEE Std 1003.1-2001, <semaphore.h>

39729 CHANGE HISTORY

39730 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39731 Issue 6

39732 The *sem_destroy()* function is marked as part of the Semaphores option.

39733 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39734 implementation does not support the Semaphores option.

39735 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/114 is applied, updating the ERRORS 2
39736 section so that the [EINVAL] error becomes optional. 2

39737 NAME

39738 sem_getvalue — get the value of a semaphore (REALTIME)

39739 SYNOPSIS

39740 SEM #include <semaphore.h>

39741 int sem_getvalue(sem_t *restrict sem, int *restrict sval);

39742

39743 DESCRIPTION

39744 The *sem_getvalue()* function shall update the location referenced by the *sval* argument to have
39745 the value of the semaphore referenced by *sem* without affecting the state of the semaphore. The
39746 updated value represents an actual semaphore value that occurred at some unspecified time
39747 during the call, but it need not be the actual value of the semaphore when it is returned to the
39748 calling process.

39749 If *sem* is locked, then the object to which *sval* points shall either be set to zero or to a negative 1
39750 number whose absolute value represents the number of processes waiting for the semaphore at 1
39751 some unspecified time during the call. 1

39752 RETURN VALUE

39753 Upon successful completion, the *sem_getvalue()* function shall return a value of zero. Otherwise,
39754 it shall return a value of -1 and set *errno* to indicate the error.

39755 ERRORS

39756 The *sem_getvalue()* function may fail if:

2

39757 [EINVAL] The *sem* argument does not refer to a valid semaphore.

39758 EXAMPLES

39759 None.

39760 APPLICATION USAGE

39761 The *sem_getvalue()* function is part of the Semaphores option and need not be available on all
39762 implementations.

39763 RATIONALE

39764 None.

39765 FUTURE DIRECTIONS

39766 None.

39767 SEE ALSO

39768 *semctl()*, *semget()*, *semop()*, *sem_post()*, *sem_timedwait()*, *sem_trywait()*, *sem_wait()*, the Base
39769 Definitions volume of IEEE Std 1003.1-2001, <semaphore.h>

39770 CHANGE HISTORY

39771 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39772 Issue 6

39773 The *sem_getvalue()* function is marked as part of the Semaphores option.

39774 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39775 implementation does not support the Semaphores option.

39776 The *sem_timedwait()* function is added to the SEE ALSO section for alignment with
39777 IEEE Std 1003.1d-1999.

39778 The **restrict** keyword is added to the *sem_getvalue()* prototype for alignment with the
39779 ISO/IEC 9899:1999 standard.

39780	IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/54 is applied.	1
39781	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/115 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.	2
39782		2

39783 NAME

39784 sem_init — initialize an unnamed semaphore (**REALTIME**)

39785 SYNOPSIS

39786 SEM #include <semaphore.h>

39787 int sem_init(sem_t *sem, int pshared, unsigned value);

39788

39789 DESCRIPTION

39790 The *sem_init()* function shall initialize the unnamed semaphore referred to by *sem*. The value of
 39791 the initialized semaphore shall be *value*. Following a successful call to *sem_init()*, the semaphore
 39792 TMO may be used in subsequent calls to *sem_wait()*, *sem_timedwait()*, *sem_trywait()*, *sem_post()*, and
 39793 *sem_destroy()*. This semaphore shall remain usable until the semaphore is destroyed. 2

39794 If the *pshared* argument has a non-zero value, then the semaphore is shared between processes;
 39795 in this case, any process that can access the semaphore *sem* can use *sem* for performing
 39796 TMO *sem_wait()*, *sem_timedwait()*, *sem_trywait()*, *sem_post()*, and *sem_destroy()* operations. 2

39797 Only *sem* itself may be used for performing synchronization. The result of referring to copies of
 39798 TMO *sem* in calls to *sem_wait()*, *sem_timedwait()*, *sem_trywait()*, *sem_post()*, and *sem_destroy()* is
 39799 undefined. 2

39800 If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any
 39801 TMO thread in this process can use *sem* for performing *sem_wait()*, *sem_timedwait()*, *sem_trywait()*,
 39802 *sem_post()*, and *sem_destroy()* operations. The use of the semaphore by threads other than those
 39803 created in the same process is undefined. 2

39804 Attempting to initialize an already initialized semaphore results in undefined behavior.

39805 RETURN VALUE

39806 Upon successful completion, the *sem_init()* function shall initialize the semaphore in *sem*.
 39807 Otherwise, it shall return -1 and set *errno* to indicate the error.

39808 ERRORS

39809 The *sem_init()* function shall fail if:

39810 [EINVAL] The *value* argument exceeds {SEM_VALUE_MAX}.

39811 [ENOSPC] A resource required to initialize the semaphore has been exhausted, or the
 39812 limit on semaphores ({SEM_NSEMS_MAX}) has been reached.

39813 [EPERM] The process lacks the appropriate privileges to initialize the semaphore.

39814 EXAMPLES

39815 None.

39816 APPLICATION USAGE

39817 The *sem_init()* function is part of the Semaphores option and need not be available on all
 39818 implementations.

39819 RATIONALE

39820 Although this volume of IEEE Std 1003.1-2001 fails to specify a successful return value, it is
 39821 likely that a later version may require the implementation to return a value of zero if the call to
 39822 *sem_init()* is successful.

39823 FUTURE DIRECTIONS

39824 None.

39825 SEE ALSO

39826 *sem_destroy()*, *sem_post()*, *sem_timedwait()*, *sem_trywait()*, *sem_wait()*, the Base Definitions
39827 volume of IEEE Std 1003.1-2001, <semaphore.h>

39828 CHANGE HISTORY

39829 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39830 Issue 6

39831 The *sem_init()* function is marked as part of the Semaphores option.

39832 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39833 implementation does not support the Semaphores option.

39834 The *sem_timedwait()* function is added to the SEE ALSO section for alignment with
39835 IEEE Std 1003.1d-1999.

39836 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/116 is applied, updating the 2
39837 DESCRIPTION to add the *sem_timedwait()* function for alignment with IEEE Std 1003.1d-1999. 2

39838 NAME

39839 sem_open — initialize and open a named semaphore (**REALTIME**)

39840 SYNOPSIS

39841 SEM #include <semaphore.h>

39842 sem_t *sem_open(const char *name, int oflag, ...);

39843

39844 DESCRIPTION

39845 The *sem_open()* function shall establish a connection between a named semaphore and a process.
 39846 Following a call to *sem_open()* with semaphore name *name*, the process may reference the
 39847 semaphore associated with *name* using the address returned from the call. This semaphore may
 39848 TMO be used in subsequent calls to *sem_wait()*, *sem_timedwait()*, *sem_trywait()*, *sem_post()*, and
 39849 *sem_close()*. The semaphore remains usable by this process until the semaphore is closed by a
 39850 successful call to *sem_close()*, *_exit()*, or one of the *exec* functions.

2

39851 The *oflag* argument controls whether the semaphore is created or merely accessed by the call to
 39852 *sem_open()*. The following flag bits may be set in *oflag*:

39853 O_CREAT This flag is used to create a semaphore if it does not already exist. If O_CREAT is
 39854 set and the semaphore already exists, then O_CREAT has no effect, except as noted
 39855 under O_EXCL. Otherwise, *sem_open()* creates a named semaphore. The O_CREAT
 39856 flag requires a third and a fourth argument: *mode*, which is of type **mode_t**, and
 39857 *value*, which is of type **unsigned**. The semaphore is created with an initial value of
 39858 *value*. Valid initial values for semaphores are less than or equal to
 39859 {SEM_VALUE_MAX}.

39860 The user ID of the semaphore is set to the effective user ID of the process; the
 39861 group ID of the semaphore is set to a system default group ID or to the effective
 39862 group ID of the process. The permission bits of the semaphore are set to the value
 39863 of the *mode* argument except those set in the file mode creation mask of the
 39864 process. When bits in *mode* other than the file permission bits are specified, the
 39865 effect is unspecified.

39866 After the semaphore named *name* has been created by *sem_open()* with the
 39867 O_CREAT flag, other processes can connect to the semaphore by calling
 39868 *sem_open()* with the same value of *name*.

39869 O_EXCL If O_EXCL and O_CREAT are set, *sem_open()* fails if the semaphore *name* exists.
 39870 The check for the existence of the semaphore and the creation of the semaphore if
 39871 it does not exist are atomic with respect to other processes executing *sem_open()*
 39872 with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the
 39873 effect is undefined.

39874 If flags other than O_CREAT and O_EXCL are specified in the *oflag* parameter, the
 39875 effect is unspecified.

39876 The *name* argument points to a string naming a semaphore object. It is unspecified whether the
 39877 name appears in the file system and is visible to functions that take pathnames as arguments.
 39878 The *name* argument conforms to the construction rules for a pathname. If *name* begins with the
 39879 slash character, then processes calling *sem_open()* with the same value of *name* shall refer to the
 39880 same semaphore object, as long as that name has not been removed. If *name* does not begin with
 39881 the slash character, the effect is implementation-defined. The interpretation of slash characters
 39882 other than the leading slash character in *name* is implementation-defined.

39883 If a process makes multiple successful calls to *sem_open()* with the same value for *name*, the
 39884 same semaphore address shall be returned for each such successful call, provided that there

39885 have been no calls to *sem_unlink()* for this semaphore, and at least one previous successful 2
39886 *sem_open()* call for this semaphore has not been matched with a *sem_close()* call. 2
39887 References to copies of the semaphore produce undefined results.

39888 RETURN VALUE

39889 Upon successful completion, the *sem_open()* function shall return the address of the semaphore.
39890 Otherwise, it shall return a value of SEM_FAILED and set *errno* to indicate the error. The symbol
39891 SEM_FAILED is defined in the <semaphore.h> header. No successful return from *sem_open()*
39892 shall return the value SEM_FAILED.

39893 ERRORS

39894 If any of the following conditions occur, the *sem_open()* function shall return SEM_FAILED and
39895 set *errno* to the corresponding value:

39896 [EACCES]	The named semaphore exists and the permissions specified by <i>oflag</i> are 39897 denied, or the named semaphore does not exist and permission to create the 39898 named semaphore is denied.
39899 [EEXIST]	O_CREAT and O_EXCL are set and the named semaphore already exists.
39900 [EINTR]	The <i>sem_open()</i> operation was interrupted by a signal.
39901 [EINVAL]	The <i>sem_open()</i> operation is not supported for the given name, or O_CREAT 39902 was specified in <i>oflag</i> and <i>value</i> was greater than {SEM_VALUE_MAX}.
39903 [EMFILE]	Too many semaphore descriptors or file descriptors are currently in use by 39904 this process.
39905 [ENAMETOOLONG]	The length of the <i>name</i> argument exceeds {PATH_MAX} or a pathname 39906 component is longer than {NAME_MAX}.
39908 [ENFILE]	Too many semaphores are currently open in the system.
39909 [ENOENT]	O_CREAT is not set and the named semaphore does not exist.
39910 [ENOSPC]	There is insufficient space for the creation of the new named semaphore.

39911 EXAMPLES

39912 None.

39913 APPLICATION USAGE

39914 The *sem_open()* function is part of the Semaphores option and need not be available on all
39915 implementations.

39916 RATIONALE

39917 Early drafts required an error return value of -1 with the type **sem_t** * for the *sem_open()*
39918 function, which is not guaranteed to be portable across implementations. The revised text
39919 provides the symbolic error code SEM_FAILED to eliminate the type conflict.

39920 FUTURE DIRECTIONS

39921 None.

39922 SEE ALSO

39923 *semctl()*, *semget()*, *semop()*, *sem_close()*, *sem_post()*, *sem_timedwait()*, *sem_trywait()*, *sem_unlink()*,
39924 *sem_wait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <semaphore.h>

39925 CHANGE HISTORY

39926 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39927 Issue 6

39928 The *sem_open()* function is marked as part of the Semaphores option.

39929 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39930 implementation does not support the Semaphores option.

39931 The *sem_timedwait()* function is added to the SEE ALSO section for alignment with
39932 IEEE Std 1003.1d-1999.

39933 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/117 is applied, updating the 2
39934 DESCRIPTION to add the *sem_timedwait()* function for alignment with IEEE Std 1003.1d-1999. 2

39935 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/118 is applied, updating the 2
39936 DESCRIPTION to describe the conditions to return the same semaphore address on a call to 2
39937 *sem_open()*. The words “and at least one previous successful *sem_open()* call for this semaphore 2
39938 has not been matched with a *sem_close()* call” are added. 2

39939 NAME

39940 sem_post — unlock a semaphore (**REALTIME**)

39941 SYNOPSIS

39942 SEM #include <semaphore.h>

39943 int sem_post(sem_t *sem);

39944

39945 DESCRIPTION

39946 The *sem_post()* function shall unlock the semaphore referenced by *sem* by performing a
39947 semaphore unlock operation on that semaphore.

39948 If the semaphore value resulting from this operation is positive, then no threads were blocked
39949 waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

39950 If the value of the semaphore resulting from this operation is zero, then one of the threads
39951 blocked waiting for the semaphore shall be allowed to return successfully from its call to
39952 *sem_wait()*. If the Process Scheduling option is supported, the thread to be unblocked shall be
39953 chosen in a manner appropriate to the scheduling policies and parameters in effect for the
39954 blocked threads. In the case of the schedulers SCHED_FIFO and SCHED_RR, the highest
39955 priority waiting thread shall be unblocked, and if there is more than one highest priority thread
39956 blocked waiting for the semaphore, then the highest priority thread that has been waiting the
39957 longest shall be unblocked. If the Process Scheduling option is not defined, the choice of a thread
39958 to unblock is unspecified.

39959 SS If the Process Sporadic Server option is supported, and the scheduling policy is
39960 SCHED_SPORADIC, the semantics are as per SCHED_FIFO above.

39961 The *sem_post()* function shall be reentrant with respect to signals and may be invoked from a
39962 signal-catching function.

39963 RETURN VALUE

39964 If successful, the *sem_post()* function shall return zero; otherwise, the function shall return -1
39965 and set *errno* to indicate the error.

39966 ERRORS

39967 The *sem_post()* function may fail if:

2

39968 [EINVAL] The *sem* argument does not refer to a valid semaphore.

39969 EXAMPLES

39970 None.

39971 APPLICATION USAGE

39972 The *sem_post()* function is part of the Semaphores option and need not be available on all
39973 implementations.

39974 RATIONALE

39975 None.

39976 FUTURE DIRECTIONS

39977 None.

39978 SEE ALSO

39979 *semctl()*, *semget()*, *semop()*, *sem_timedwait()*, *sem_trywait()*, *sem_wait()*, the Base Definitions
39980 volume of IEEE Std 1003.1-2001, <semaphore.h>

39981 **CHANGE HISTORY**

39982 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

39983 **Issue 6**39984 The *sem_post()* function is marked as part of the Semaphores option.39985 The [ENOSYS] error condition has been removed as stubs need not be provided if an
39986 implementation does not support the Semaphores option.39987 The *sem_timedwait()* function is added to the SEE ALSO section for alignment with
39988 IEEE Std 1003.1d-1999.39989 SCED_SPORADIC is added to the list of scheduling policies for which the thread that is to be
39990 unblocked is specified for alignment with IEEE Std 1003.1d-1999.39991 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/119 is applied, updating the ERRORS 2
39992 section so that the [EINVAL] error becomes optional. 2

39993 NAME

39994 sem_timedwait — lock a semaphore (ADVANCED REALTIME)

39995 SYNOPSIS

```
39996 SEM TMO #include <semaphore.h>
39997 #include <time.h>
39998 int sem_timedwait(sem_t *restrict sem,
39999     const struct timespec *restrict abs_timeout);
```

40001 DESCRIPTION

40002 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.

40006 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.

40010 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.

40014 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.

40017 RETURN VALUE

40018 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.

40022 ERRORS

40023 The *sem_timedwait()* function shall fail if:

40024 [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.

40027 [ETIMEDOUT] The semaphore could not be locked before the specified timeout expired.

40028 The *sem_timedwait()* function may fail if:

40029 [EDEADLK] A deadlock condition was detected.

40030 [EINTR] A signal interrupted this function.

40031 [EINVAL] The *sem* argument does not refer to a valid semaphore.

2

40032 EXAMPLES

40033 None.

40034 APPLICATION USAGE

40035 Applications using these functions may be subject to priority inversion, as discussed in the Base
40036 Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

40037 The *sem_timedwait()* function is part of the Semaphores and Timeouts options and need not be
40038 provided on all implementations.

40039 RATIONALE

40040 None.

40041 FUTURE DIRECTIONS

40042 None.

40043 SEE ALSO

40044 *sem_post()*, *sem_trywait()*, *sem_wait()*, *semctl()*, *semget()*, *semop()*, *time()*, the Base Definitions
40045 volume of IEEE Std 1003.1-2001, <semaphore.h>, <time.h>

40046 CHANGE HISTORY

40047 First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

40048 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/120 is applied, updating the ERRORS 2
40049 section so that the [EINVAL] error becomes optional. 2

40050 NAME

40051 sem_trywait, sem_wait — lock a semaphore (**REALTIME**)

40052 SYNOPSIS

40053 SEM #include <semaphore.h>

40054 int sem_trywait(sem_t *sem);

40055 int sem_wait(sem_t *sem);

40056

40057 DESCRIPTION

40058 The *sem_trywait()* function shall lock the semaphore referenced by *sem* only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it shall not lock the semaphore.

40061 The *sem_wait()* function shall lock the semaphore referenced by *sem* by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread shall not return from the call to *sem_wait()* until it either locks the semaphore or the call is interrupted by a signal.

40065 Upon successful return, the state of the semaphore shall be locked and shall remain locked until the *sem_post()* function is executed and returns successfully.

40067 The *sem_wait()* function is interruptible by the delivery of a signal.

40068 RETURN VALUE

40069 The *sem_trywait()* and *sem_wait()* functions 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.

40073 ERRORS

40074 The *sem_trywait()* and *sem_wait()* functions shall fail if:

40075 [EAGAIN] The semaphore was already locked, so it cannot be immediately locked by the *sem_trywait()* operation (*sem_trywait()* only).

40077 The *sem_trywait()* and *sem_wait()* functions may fail if:

40078 [EDEADLK] A deadlock condition was detected.

40079 [EINTR] A signal interrupted this function.

40080 [EINVAL] The *sem* argument does not refer to a valid semaphore.

2

40081 EXAMPLES

40082 None.

40083 APPLICATION USAGE

40084 Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.285, Priority Inversion.

40086 The *sem_trywait()* and *sem_wait()* functions are part of the Semaphores option and need not be provided on all implementations.

40088 RATIONALE

40089 None.

40090 FUTURE DIRECTIONS

40091 None.

40092 SEE ALSO

40093 *semctl()*, *semget()*, *semop()*, *sem_post()*, *sem_timedwait()*, the Base Definitions volume of
40094 IEEE Std 1003.1-2001, <semaphore.h>

40095 CHANGE HISTORY

40096 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

40097 Issue 6

40098 The *sem_trywait()* and *sem_wait()* functions are marked as part of the Semaphores option.

40099 The [ENOSYS] error condition has been removed as stubs need not be provided if an
40100 implementation does not support the Semaphores option.

40101 The *sem_timedwait()* function is added to the SEE ALSO section for alignment with
40102 IEEE Std 1003.1d-1999.

40103 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/121 is applied, updating the ERRORS 2
40104 section so that the [EINVAL] error becomes optional. 2

40105 **NAME**

40106 `sem_unlink` — remove a named semaphore (**REALTIME**)

40107 **SYNOPSIS**

40108 `SEM #include <semaphore.h>`

40109 `int sem_unlink(const char *name);`

40110

40111 **DESCRIPTION**

40112 The `sem_unlink()` function shall remove the semaphore named by the string *name*. If the
40113 semaphore named by *name* is currently referenced by other processes, then `sem_unlink()` shall
40114 have no effect on the state of the semaphore. If one or more processes have the semaphore open
40115 when `sem_unlink()` is called, destruction of the semaphore is postponed until all references to the
40116 semaphore have been destroyed by calls to `sem_close()`, `_exit()`, or `exec`. Calls to `sem_open()` to
40117 recreate or reconnect to the semaphore refer to a new semaphore after `sem_unlink()` is called. The
40118 `sem_unlink()` call shall not block until all references have been destroyed; it shall return
40119 immediately.

40120 **RETURN VALUE**

40121 Upon successful completion, the `sem_unlink()` function shall return a value of 0. Otherwise, the
40122 semaphore shall not be changed and the function shall return a value of -1 and set `errno` to
40123 indicate the error.

40124 **ERRORS**

40125 The `sem_unlink()` function shall fail if:

40126 [EACCES] Permission is denied to unlink the named semaphore.

40127 [ENAMETOOLONG] The length of the *name* argument exceeds {PATH_MAX} or a pathname
40128 component is longer than {NAME_MAX}.

40130 [ENOENT] The named semaphore does not exist.

40131 **EXAMPLES**

40132 None.

40133 **APPLICATION USAGE**

40134 The `sem_unlink()` function is part of the Semaphores option and need not be available on all
40135 implementations.

40136 **RATIONALE**

40137 None.

40138 **FUTURE DIRECTIONS**

40139 None.

40140 **SEE ALSO**

40141 `semctl()`, `semget()`, `semop()`, `sem_close()`, `sem_open()`, the Base Definitions volume of
40142 IEEE Std 1003.1-2001, `<semaphore.h>`

40143 **CHANGE HISTORY**

40144 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

40145 **Issue 6**

40146 The `sem_unlink()` function is marked as part of the Semaphores option.

40147
40148

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

40149 NAME

40150 sem_wait — lock a semaphore (**REALTIME**)

40151 SYNOPSIS

40152 SEM #include <semaphore.h>

40153 int sem_wait(sem_t *sem);

40154

40155 DESCRIPTION

40156 Refer to *sem_trywait()*.

40157 NAME

40158 semctl — XSI semaphore control operations

40159 SYNOPSIS

```
40160 XSI #include <sys/sem.h>
40161 int semctl(int semid, int semnum, int cmd, ...);
40162
```

40163 DESCRIPTION

40164 The *semctl()* function operates on XSI semaphores (see the Base Definitions volume of
 40165 IEEE Std 1003.1-2001, Section 4.15, Semaphore). It is unspecified whether this function
 40166 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 40167 page 41).

40168 The *semctl()* function provides a variety of semaphore control operations as specified by *cmd*.
 40169 The fourth argument is optional and depends upon the operation requested. If required, it is of
 40170 type **union semun**, which the application shall explicitly declare:

```
40171 union semun {
40172     int val;
40173     struct semid_ds *buf;
40174     unsigned short  *array;
40175 } arg;
```

40176 The following semaphore control operations as specified by *cmd* are executed with respect to the
 40177 semaphore specified by *semid* and *semnum*. The level of permission required for each operation
 40178 is shown with each command; see Section 2.7 (on page 39). The symbolic names for the values
 40179 of *cmd* are defined in the <sys/sem.h> header:

40180	GETVAL	Return the value of <i>semval</i> ; see <sys/sem.h>. Requires read permission.
40181	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).
40185	GETPID	Return the value of <i>sempid</i> . Requires read permission.
40186	GETNCNT	Return the value of <i>semncnt</i> . Requires read permission.
40187	GETZCNT	Return the value of <i>semzcnt</i> . Requires read permission.
40188	The following values of <i>cmd</i> operate on each <i>semval</i> in the set of semaphores:	
40189	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.
40192	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.
40197	The following values of <i>cmd</i> are also available:	
40198	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

40201		<sys/sem.h>. Requires read permission.
40202	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> :
40203		<i>sem_perm.uid</i>
40204		<i>sem_perm.gid</i>
40205		<i>sem_perm.mode</i>
40206		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.
40207		
40208		
40209		
40210		
40211		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> .
40212		
40213		
40214		
40215	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> .
40216		
40217		
40218		
40219		
40220		
40221	RETURN VALUE	
40222		If successful, the value returned by <i>semctl()</i> depends on <i>cmd</i> as follows:
40223	GETVAL	The value of <i>semval</i> .
40224	GETPID	The value of <i>sempid</i> .
40225	GETNCNT	The value of <i>semncnt</i> .
40226	GETZCNT	The value of <i>semzcnt</i> .
40227	All others	0.
40228		Otherwise, <i>semctl()</i> shall return -1 and set <i>errno</i> to indicate the error.
40229	ERRORS	
40230		The <i>semctl()</i> function shall fail if:
40231	[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 39).
40232		
40233	[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.
40234		
40235		
40236	[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> .
40237		
40238		
40239		
40240	[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.
40241		

40242 EXAMPLES

40243 None.

40244 APPLICATION USAGE

40245 The fourth parameter in the SYNOPSIS section is now specified as " . . ." in order to avoid a
40246 clash with the ISO C standard when referring to the union *semun* (as defined in Issue 3) and for
40247 backwards-compatibility.

40248 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
40249 Application developers who need to use IPC should design their applications so that modules
40250 using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the
40251 alternative interfaces.

40252 RATIONALE

40253 None.

40254 FUTURE DIRECTIONS

40255 None.

40256 SEE ALSO

40257 Section 2.7 (on page 39), Section 2.8 (on page 41), *semget()*, *semop()*, *sem_close()*, *sem_destroy()*,
40258 *sem_getvalue()*, *sem_init()*, *sem_open()*, *sem_post()*, *sem_unlink()*, *sem_wait()*, the Base Definitions
40259 volume of IEEE Std 1003.1-2001, <sys/sem.h>

40260 CHANGE HISTORY

40261 First released in Issue 2. Derived from Issue 2 of the SVID.

40262 Issue 5

40263 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
40264 DIRECTIONS to the APPLICATION USAGE section.

40265 NAME

40266 semget — get set of XSI semaphores

40267 SYNOPSIS

40268 XSI #include <sys/sem.h>

40269 int semget(key_t key, int nsems, int semflg);

40270

40271 DESCRIPTION

40272 The *semget()* function operates on XSI semaphores (see the Base Definitions volume of
 40273 IEEE Std 1003.1-2001, Section 4.15, Semaphore). It is unspecified whether this function
 40274 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 40275 page 41).

40276 The *semget()* function shall return the semaphore identifier associated with *key*.

40277 A semaphore identifier with its associated **semid_ds** data structure and its associated set of
 40278 *nsems* semaphores (see <sys/sem.h>) is created for *key* if one of the following is true:

- 40279 • The argument *key* is equal to IPC_PRIVATE.
- 40280 • The argument *key* does not already have a semaphore identifier associated with it and (*semflg*
 40281 &IPC_CREAT) is non-zero.

40282 Upon creation, the **semid_ds** data structure associated with the new semaphore identifier is
 40283 initialized as follows:

- 40284 • In the operation permissions structure *sem_perm.cuid*, *sem_perm.uid*, *sem_perm.cgid*, and
 40285 *sem_perm.gid* shall be set equal to the effective user ID and effective group ID, respectively, of
 40286 the calling process.
- 40287 • The low-order 9 bits of *sem_perm.mode* shall be set equal to the low-order 9 bits of *semflg*.
- 40288 • The variable *sem_nsems* shall be set equal to the value of *nsems*.
- 40289 • The variable *sem_otime* shall be set equal to 0 and *sem_ctime* shall be set equal to the current
 40290 time.
- 40291 • The data structure associated with each semaphore in the set need not be initialized. The
 40292 *semctl()* function with the command SETVAL or SETALL can be used to initialize each
 40293 semaphore.

40294 RETURN VALUE

40295 Upon successful completion, *semget()* shall return a non-negative integer, namely a semaphore
 40296 identifier; otherwise, it shall return -1 and set *errno* to indicate the error.

40297 ERRORS

40298 The *semget()* function shall fail if:

- | | |
|----------------|---|
| 40299 [EACCES] | A semaphore identifier exists for <i>key</i> , but operation permission as specified by
40300 the low-order 9 bits of <i>semflg</i> would not be granted; see Section 2.7 (on page
40301 39). |
| 40302 [EEXIST] | A semaphore identifier exists for the argument <i>key</i> but ((<i>semflg</i> &IPC_CREAT)
40303 &&(<i>semflg</i> &IPC_EXCL)) is non-zero. |
| 40304 [EINVAL] | The value of <i>nsems</i> is either less than or equal to 0 or greater than the system-
40305 imposed limit, or a semaphore identifier exists for the argument <i>key</i> , but the
40306 number of semaphores in the set associated with it is less than <i>nsems</i> and
40307 <i>nsems</i> is not equal to 0. |

40308	[ENOENT]	A semaphore identifier does not exist for the argument <i>key</i> and (<i>semflg</i> &IPC_CREAT) is equal to 0.
40310	[ENOSPC]	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system-wide would be exceeded.

40312 EXAMPLES

40313 Creating a Semaphore Identifier

40314 The following example gets a unique semaphore key using the *ftok()* function, then gets a
40315 semaphore ID associated with that key using the *semget()* function (the first call also tests to
40316 make sure the semaphore exists). If the semaphore does not exist, the program creates it, as
40317 shown by the second call to *semget()*. In creating the semaphore for the queuing process, the
40318 program attempts to create one semaphore with read/write permission for all. It also uses the
40319 IPC_EXCL flag, which forces *semget()* to fail if the semaphore already exists.

40320 After creating the semaphore, the program uses a call to *semop()* to initialize it to the values in
40321 the *sbuf* array. The number of processes that can execute concurrently without queuing is
40322 initially set to 2. The final call to *semget()* creates a semaphore identifier that can be used later in
40323 the program.

```
40324 #include <sys/types.h>
40325 #include <stdio.h>
40326 #include <sys/ipc.h>
40327 #include <sys/sem.h>
40328 #include <sys/stat.h>
40329 #include <errno.h>
40330 #include <unistd.h>
40331 #include <stdlib.h>
40332 #include <pwd.h>
40333 #include <fcntl.h>
40334 #include <limits.h>
40335 ...
40336 key_t semkey;
40337 int semid, pfd, fv;
40338 struct sembuf sbuf;
40339 char *lgn;
40340 char filename[PATH_MAX+1];
40341 struct stat outstat;
40342 struct passwd *pw;
40343 ...
40344 /* Get unique key for semaphore. */
40345 if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
40346     perror("IPC error: ftok");
40347     exit(1);
40348 }
40349 /* Get semaphore ID associated with this key. */
40350 if ((semid = semget(semkey, 0, 0)) == -1) {
40351     /* Semaphore does not exist - Create. */
40352     if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
40353         S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
40354         /* Initialize the semaphore. */
40355         sbuf.sem_num = 0;
```

```

40356         sbuf.sem_op = 2; /* This is the number of runs
40357                     without queuing. */
40358         sbuf.sem_flg = 0;
40359         if (semop(semid, &sbuf, 1) == -1) {
40360             perror("IPC error: semop"); exit(1);
40361         }
40362     }
40363     else if (errno == EEXIST) {
40364         if ((semid = semget(semkey, 0, 0)) == -1) {
40365             perror("IPC error 1: semget"); exit(1);
40366         }
40367     }
40368     else {
40369         perror("IPC error 2: semget"); exit(1);
40370     }
40371 }
40372 ...

```

40373 APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the alternative interfaces.

40378 RATIONALE

40379 None.

40380 FUTURE DIRECTIONS

40381 None.

40382 SEE ALSO

40383 Section 2.7 (on page 39), Section 2.8 (on page 41), *semctl()*, *semop()*, *sem_close()*, *sem_destroy()*,
 40384 *sem_getvalue()*, *sem_init()*, *sem_open()*, *sem_post()*, *sem_unlink()*, *sem_wait()*, the Base Definitions
 40385 volume of IEEE Std 1003.1-2001, <sys/sem.h>

40386 CHANGE HISTORY

40387 First released in Issue 2. Derived from Issue 2 of the SVID.

40388 Issue 5

40389 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
 40390 DIRECTIONS to a new APPLICATION USAGE section.

40391 Issue 6

40392 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/122 is applied, updating the 40393 DESCRIPTION from “each semaphore in the set shall not be initialized” to “each semaphore in 40394 the set need not be initialized”.	2 2 2 2
---	------------------

40395 NAME

40396 semop — XSI semaphore operations

40397 SYNOPSIS

40398 XSI #include <sys/sem.h>

40399 int semop(int *semid*, struct sembuf **sops*, size_t *nsops*);

40400

40401 DESCRIPTION

40402 The *semop()* function operates on XSI semaphores (see the Base Definitions volume of
 40403 IEEE Std 1003.1-2001, Section 4.15, Semaphore). It is unspecified whether this function
 40404 interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
 40405 page 41).

40406 The *semop()* function shall perform atomically a user-defined array of semaphore operations on
 40407 the set of semaphores associated with the semaphore identifier specified by the argument *semid*.

40408 The argument *sops* is a pointer to a user-defined array of semaphore operation structures. The
 40409 implementation shall not modify elements of this array unless the application uses
 40410 implementation-defined extensions.

40411 The argument *nsops* is the number of such structures in the array.

40412 Each structure, **sembuf**, includes the following members:

40413

40414

40415

40416

40417

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.

40418 Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore
 40419 specified by *semid* and *sem_num*.

40420 The variable *sem_op* specifies one of three semaphore operations:

- 40421 1. If *sem_op* is a negative integer and the calling process has alter permission, one of the
 40422 following shall occur:

- 40423 • If *semval*(see <sys/sem.h>) is greater than or equal to the absolute value of *sem_op*, the
 40424 absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg* &SEM_UNDO) is
 40425 non-zero, the absolute value of *sem_op* shall be added to the calling process' *semadj*
 40426 value for the specified semaphore.

- 40427 • If *semval* is less than the absolute value of *sem_op* and (*sem_flg* &IPC_NOWAIT) is non-
 40428 zero, *semop()* shall return immediately.

- 40429 • If *semval* is less than the absolute value of *sem_op* and (*sem_flg* &IPC_NOWAIT) is 0,
 40430 *semop()* shall increment the *semncnt* associated with the specified semaphore and
 40431 suspend execution of the calling thread until one of the following conditions occurs:

- 40432 — The value of *semval* becomes greater than or equal to the absolute value of *sem_op*.
 40433 When this occurs, the value of *semncnt* associated with the specified semaphore
 40434 shall be decremented, the absolute value of *sem_op* shall be subtracted from *semval*
 40435 and, if (*sem_flg* &SEM_UNDO) is non-zero, the absolute value of *sem_op* shall be
 40436 added to the calling process' *semadj* value for the specified semaphore.

- 40437 — The *semid* for which the calling thread is awaiting action is removed from the
 40438 system. When this occurs, *errno* shall be set equal to [EIDRM] and -1 shall be

40439 returned.

40440 — The calling thread receives a signal that is to be caught. When this occurs, the value
40441 of *semncnt* associated with the specified semaphore shall be decremented, and the
40442 calling thread shall resume execution in the manner prescribed in *sigaction()*.

40443 2. If *sem_op* is a positive integer and the calling process has alter permission, the value of
40444 *sem_op* shall be added to *semval* and, if (*sem_flg* & SEM_UNDO) is non-zero, the value of
40445 *sem_op* shall be subtracted from the calling process' *semadj* value for the specified
40446 semaphore.

40447 3. If *sem_op* is 0 and the calling process has read permission, one of the following shall occur:

40448 • If *semval* is 0, *semop()* shall return immediately.

40449 • If *semval* is non-zero and (*sem_flg* & IPC_NOWAIT) is non-zero, *semop()* shall return
40450 immediately.

40451 • If *semval* is non-zero and (*sem_flg* & IPC_NOWAIT) is 0, *semop()* shall increment the
40452 *semzcnt* associated with the specified semaphore and suspend execution of the calling
40453 thread until one of the following occurs:

40454 — The value of *semval* becomes 0, at which time the value of *semzcnt* associated with
40455 the specified semaphore shall be decremented.

40456 — The *semid* for which the calling thread is awaiting action is removed from the
40457 system. When this occurs, *errno* shall be set equal to [EIDRM] and -1 shall be
40458 returned.

40459 — The calling thread receives a signal that is to be caught. When this occurs, the value
40460 of *semzcnt* associated with the specified semaphore shall be decremented, and the
40461 calling thread shall resume execution in the manner prescribed in *sigaction()*.

Upon successful completion, the value of *semid* for each semaphore specified in the array pointed to by *sops* shall be set equal to the process ID of the calling process.

40464 RETURN VALUE

Upon successful completion, *semop()* shall return 0; otherwise, it shall return -1 and set *errno* to indicate the error.

40467 ERRORS

40468 The *semop()* function shall fail if:

40469	[E2BIG]	The value of <i>nsops</i> is greater than the system-imposed maximum.
40470	[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 39).
40471		
40472	[EAGAIN]	The operation would result in suspension of the calling process but (<i>sem_flg</i> &IPC_NOWAIT) is non-zero.
40473		
40474	[EFBIG]	The value of <i>sem_num</i> is less than 0 or greater than or equal to the number of semaphores in the set associated with <i>semid</i> .
40475		
40476	[EIDRM]	The semaphore identifier <i>semid</i> is removed from the system.
40477	[EINTR]	The <i>semop()</i> function was interrupted by a signal.
40478	[EINVAL]	The value of <i>semid</i> is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a SEM_UNDO would exceed the system-imposed limit.
40479		
40480		

40481	[ENOSPC]	The limit on the number of individual processes requesting a SEM_UNDO would be exceeded.
40483	[ERANGE]	An operation would cause a <i>semval</i> to overflow the system-imposed limit, or an operation would cause a <i>semadj</i> value to overflow the system-imposed limit.

40486 EXAMPLES

40487 Setting Values in Semaphores

40488 The following example sets the values of the two semaphores associated with the *semid* identifier to the values contained in the *sb* array.

```
40490 #include <sys/sem.h>
40491 ...
40492 int semid;
40493 struct sembuf sb[2];
40494 int nsops = 2;
40495 int result;

40496 /* Adjust value of semaphore in the semaphore array semid. */
40497 sb[0].sem_num = 0;
40498 sb[0].sem_op = -1;
40499 sb[0].sem_flg = SEM_UNDO | IPC_NOWAIT;
40500 sb[1].sem_num = 1;
40501 sb[1].sem_op = 1;
40502 sb[1].sem_flg = 0;

40503 result = semop(semid, sb, nsops);
```

40504 Creating a Semaphore Identifier

40505 The following example gets a unique semaphore key using the *ftok()* function, then gets a
40506 semaphore ID associated with that key using the *semget()* function (the first call also tests to
40507 make sure the semaphore exists). If the semaphore does not exist, the program creates it, as
40508 shown by the second call to *semget()*. In creating the semaphore for the queuing process, the
40509 program attempts to create one semaphore with read/write permission for all. It also uses the
40510 IPC_EXCL flag, which forces *semget()* to fail if the semaphore already exists.

40511 After creating the semaphore, the program uses a call to *semop()* to initialize it to the values in
40512 the *sbuff* array. The number of processes that can execute concurrently without queuing is
40513 initially set to 2. The final call to *semget()* creates a semaphore identifier that can be used later in
40514 the program.

40515 The final call to *semop()* acquires the semaphore and waits until it is free; the SEM_UNDO
40516 option releases the semaphore when the process exits, waiting until there are less than two
40517 processes running concurrently.

```
40518 #include <sys/types.h>
40519 #include <stdio.h>
40520 #include <sys/ipc.h>
40521 #include <sys/sem.h>
40522 #include <sys/stat.h>
40523 #include <errno.h>
40524 #include <unistd.h>
40525 #include <stdlib.h>
```

```

40526     #include <pwd.h>
40527     #include <fcntl.h>
40528     #include <limits.h>
40529     ...
40530     key_t semkey;
40531     int semid, pfd, fv;
40532     struct sembuf sbuf;
40533     char *lgn;
40534     char filename[PATH_MAX+1];
40535     struct stat outstat;
40536     struct passwd *pw;
40537     ...
40538     /* Get unique key for semaphore. */
40539     if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
40540         perror("IPC error: ftok");
40541         exit(1);
40542     }
40543     /* Get semaphore ID associated with this key. */
40544     if ((semid = semget(semkey, 0, 0)) == -1) {
40545         /* Semaphore does not exist - Create. */
40546         if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
40547             S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
40548         {
40549             /* Initialize the semaphore. */
40550             sbuf.sem_num = 0;
40551             sbuf.sem_op = 2; /* This is the number of runs without queuing. */
40552             sbuf.sem_flg = 0;
40553             if (semop(semid, &sbuf, 1) == -1) {
40554                 perror("IPC error: semop");
40555                 exit(1);
40556             }
40557             else if (errno == EEXIST) {
40558                 if ((semid = semget(semkey, 0, 0)) == -1) {
40559                     perror("IPC error 1: semget");
40560                     exit(1);
40561                 }
40562                 else {
40563                     perror("IPC error 2: semget");
40564                     exit(1);
40565                 }
40566             }
40567             sbuf.sem_num = 0;
40568             sbuf.sem_op = -1;
40569             sbuf.sem_flg = SEM_UNDO;
40570             if (semop(semid, &sbuf, 1) == -1) {
40571                 perror("IPC Error: semop");
40572                 exit(1);
40573             }
40574             ...
40575             sbuf.sem_num = 0;
40576             sbuf.sem_op = -1;
40577             sbuf.sem_flg = SEM_UNDO;
40578             if (semop(semid, &sbuf, 1) == -1) {
40579                 perror("IPC Error: semop");
40580                 exit(1);
40581             }
40582         }
40583     }
40584     ...
40585     sbuf.sem_num = 0;
40586     sbuf.sem_op = -1;
40587     sbuf.sem_flg = SEM_UNDO;
40588     if (semop(semid, &sbuf, 1) == -1) {
40589         perror("IPC Error: semop");
40590         exit(1);
40591     }
40592 }
```

40572 APPLICATION USAGE

40573 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
 40574 Application developers who need to use IPC should design their applications so that modules
 40575 using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the
 40576 alternative interfaces.

40577 RATIONALE

40578 None.

40579 FUTURE DIRECTIONS

40580 None.

40581 SEE ALSO

40582 Section 2.7 (on page 39), Section 2.8 (on page 41), *exec*, *exit()*, *fork()*, *semctl()*, *semget()*,
40583 *sem_close()*, *sem_destroy()*, *sem_getvalue()*, *sem_init()*, *sem_open()*, *sem_post()*, *sem_unlink()*,
40584 *sem_wait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/ipc.h>, <sys/sem.h>,
40585 <sys/types.h>

40586 CHANGE HISTORY

40587 First released in Issue 2. Derived from Issue 2 of the SVID.

40588 Issue 5

40589 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
40590 DIRECTIONS to a new APPLICATION USAGE section.

40591 NAME

40592 send — send a message on a socket

40593 SYNOPSIS

```
40594 #include <sys/socket.h>
40595 ssize_t send(int socket, const void *buffer, size_t length, int flags);
```

40596 DESCRIPTION

40597 The *send()* function shall initiate transmission of a message from the specified socket to its peer.
40598 The *send()* function shall send a message only when the socket is connected (including when the
40599 peer of a connectionless socket has been set via *connect()*).

40600 The *send()* function takes the following arguments:

40601	<i>socket</i>	Specifies the socket file descriptor.
40602	<i>buffer</i>	Points to the buffer containing the message to send.
40603	<i>length</i>	Specifies the length of the message in bytes.
40604	<i>flags</i>	Specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:
40606	MSG_EOR	Terminates a record (if supported by the protocol).
40607	MSG_OOB	Sends out-of-band data on sockets that support out-of-band communications. The significance and semantics of out-of-band data are protocol-specific.

40610 The length of the message to be sent is specified by the *length* argument. If the message is too
40611 long to pass through the underlying protocol, *send()* shall fail and no data shall be transmitted.

40612 Successful completion of a call to *send()* does not guarantee delivery of the message. A return
40613 value of -1 indicates only locally-detected errors.

40614 If space is not available at the sending socket to hold the message to be transmitted, and the
40615 socket file descriptor does not have O_NONBLOCK set, *send()* shall block until space is
40616 available. If space is not available at the sending socket to hold the message to be transmitted,
40617 and the socket file descriptor does have O_NONBLOCK set, *send()* shall fail. The *select()* and
40618 *poll()* functions can be used to determine when it is possible to send more data.

40619 The socket in use may require the process to have appropriate privileges to use the *send()*
40620 function.

40621 RETURN VALUE

40622 Upon successful completion, *send()* shall return the number of bytes sent. Otherwise, -1 shall be
40623 returned and *errno* set to indicate the error.

40624 ERRORS

40625 The *send()* function shall fail if:

40626 [EAGAIN] or [EWOULDBLOCK]

40627 The socket's file descriptor is marked O_NONBLOCK and the requested
40628 operation would block.

40629 [EBADF] The *socket* argument is not a valid file descriptor.

40630 [ECONNRESET] A connection was forcibly closed by a peer.

40631 [EDESTADDRREQ]

40632 The socket is not connection-mode and no peer address is set.

40633	[EINTR]	A signal interrupted <i>send()</i> before any data was transmitted.
40634	[EMSGSIZE]	The message is too large to be sent all at once, as the socket requires.
40635	[ENOTCONN]	The socket is not connected or otherwise has not had the peer pre-specified.
40636	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
40637	[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> .
40638	[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.
40642		The <i>send()</i> function may fail if:
40643	[EACCES]	The calling process does not have the appropriate privileges.
40644	[EIO]	An I/O error occurred while reading from or writing to the file system.
40645	[ENETDOWN]	The local network interface used to reach the destination is down.
40646	[ENETUNREACH]	
40647		No route to the network is present.
40648	[ENOBUFS]	Insufficient resources were available in the system to perform the operation.

40649 EXAMPLES

40650 None.

40651 APPLICATION USAGE

40652 The *send()* function is equivalent to *sendto()* with a null pointer *dest_len* argument, and to *write()* if no flags are used.

40654 RATIONALE

40655 None.

40656 FUTURE DIRECTIONS

40657 None.

40658 SEE ALSO

40659 *connect()*, *getsockopt()*, *poll()*, *recv()*, *recvfrom()*, *recvmsg()*, *select()*, *sendmsg()*, *sendto()*,
40660 *setsockopt()*, *shutdown()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001,
40661 <sys/socket.h>

40662 CHANGE HISTORY

40663 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

40664 NAME

40665 sendmsg — send a message on a socket using a message structure

40666 SYNOPSIS

```
40667 #include <sys/socket.h>
40668 ssize_t sendmsg(int socket, const struct msghdr *message, int flags);
```

40669 DESCRIPTION

40670 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.

40674 The *sendmsg()* function takes the following arguments:

40675 <i>socket</i>	Specifies the socket file descriptor.
40676 <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.
40679 <i>flags</i>	Specifies the type of message transmission. The application may specify 0 or the following flag:
40681	MSG_EOR Terminates a record (if supported by the protocol).
40682	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.

40685 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.

40690 Successful completion of a call to *sendmsg()* does not guarantee delivery of the message. A return value of -1 indicates only locally-detected errors.

40692 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.

40697 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.

40699 The socket in use may require the process to have appropriate privileges to use the *sendmsg()* function.

40701 RETURN VALUE

40702 Upon successful completion, *sendmsg()* shall return the number of bytes sent. Otherwise, -1 shall be returned and *errno* set to indicate the error.

40704 ERRORS

40705 The *sendmsg()* function shall fail if:

40706 [EAGAIN] or [EWOULDBLOCK]

40707 The socket's file descriptor is marked O_NONBLOCK and the requested

40708		operation would block.
40709	[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
40710		
40711	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
40712	[ECONNRESET]	A connection was forcibly closed by a peer.
40713	[EINTR]	A signal interrupted <i>sendmsg()</i> before any data was transmitted.
40714	[EINVAL]	The sum of the <i>iov_len</i> values overflows an ssize_t .
40715	[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}.
40716		
40717		
40718	[ENOTCONN]	The socket is connection-mode but is not connected.
40719	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
40720	[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> .
40721		
40722	[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.
40723		
40724		
40725		If the address family of the socket is AF_UNIX, then <i>sendmsg()</i> shall fail if:
40726	[EIO]	An I/O error occurred while reading from or writing to the file system.
40727	[ELOOP]	A loop exists in symbolic links encountered during resolution of the pathname in the socket address.
40728		
40729	[ENAMETOOLONG]	
40730		A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
40731		
40732	[ENOENT]	A component of the pathname does not name an existing file or the path name is an empty string.
40733		
40734	[ENOTDIR]	A component of the path prefix of the pathname in the socket address is not a directory.
40735		
40736		The <i>sendmsg()</i> function may fail if:
40737	[EACCES]	Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
40738		
40739	[EDESTADDRREQ]	
40740		The socket is not connection-mode and does not have its peer address set, and no destination address was specified.
40741		
40742	[EHOSTUNREACH]	
40743		The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
40744		
40745	[EIO]	An I/O error occurred while reading from or writing to the file system.
40746	[EISCONN]	A destination address was specified and the socket is already connected.
40747	[ENETDOWN]	The local network interface used to reach the destination is down.

40748 [ENETUNREACH]
40749 No route to the network is present.

40750 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

40751 [ENOMEM] Insufficient memory was available to fulfill the request.

40752 If the address family of the socket is AF_UNIX, then *sendmsg()* may fail if:

40753 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in the socket address.

40755 [ENAMETOOLONG]
40756 Pathname resolution of a symbolic link produced an intermediate result
40757 whose length exceeds {PATH_MAX}.

40758 EXAMPLES

40759 Done.

40760 APPLICATION USAGE

40761 The *select()* and *poll()* functions can be used to determine when it is possible to send more data.

40762 RATIONALE

40763 None.

40764 FUTURE DIRECTIONS

40765 None.

40766 SEE ALSO

40767 *getsockopt()*, *poll()*, *recv()*, *recvfrom()*, *recvmsg()*, *select()*, *send()*, *sendto()*, *setsockopt()*,
40768 *shutdown()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

40769 CHANGE HISTORY

40770 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

40771 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
40772 [ELOOP] error condition is added.

40773 NAME

40774 sendto — send a message on a socket

40775 SYNOPSIS

```
40776 #include <sys/socket.h>
40777 ssize_t sendto(int socket, const void *message, size_t length,
40778           int flags, const struct sockaddr *dest_addr,
40779           socklen_t dest_len);
```

40780 DESCRIPTION

40781 The *sendto()* 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 *dest_addr*. If the socket is connection-mode, *dest_addr* shall be ignored.

40784 The *sendto()* function takes the following arguments:

40785 <i>socket</i>	Specifies the socket file descriptor.
40786 <i>message</i>	Points to a buffer containing the message to be sent.
40787 <i>length</i>	Specifies the size of the message in bytes.
40788 <i>flags</i>	Specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:
40790	MSG_EOR Terminates a record (if supported by the protocol).
40791	MSG_OOB Sends out-of-band data on sockets that support out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
40794 <i>dest_addr</i>	Points to a sockaddr structure containing the destination address. The length and format of the address depend on the address family of the socket.
40796 <i>dest_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>dest_addr</i> argument.

40798 If the socket protocol supports broadcast and the specified address is a broadcast address for the
40799 socket protocol, *sendto()* shall fail if the SO_BROADCAST option is not set for the socket.

40800 The *dest_addr* argument specifies the address of the target. The *length* argument specifies the
40801 length of the message.

40802 Successful completion of a call to *sendto()* does not guarantee delivery of the message. A return
40803 value of -1 indicates only locally-detected errors.

40804 If space is not available at the sending socket to hold the message to be transmitted and the
40805 socket file descriptor does not have O_NONBLOCK set, *sendto()* shall block until space is
40806 available. If space is not available at the sending socket to hold the message to be transmitted
40807 and the socket file descriptor does have O_NONBLOCK set, *sendto()* shall fail.

40808 The socket in use may require the process to have appropriate privileges to use the *sendto()*
40809 function.

40810 RETURN VALUE

40811 Upon successful completion, *sendto()* shall return the number of bytes sent. Otherwise, -1 shall
40812 be returned and *errno* set to indicate the error.

40813 ERRORS

- 40814 The *sendto()* function shall fail if:
- 40815 [EAFNOSUPPORT]
40816 Addresses in the specified address family cannot be used with this socket.
- 40817 [EAGAIN] or [EWOULDBLOCK]
40818 The socket's file descriptor is marked O_NONBLOCK and the requested
40819 operation would block.
- 40820 [EBADF] The *socket* argument is not a valid file descriptor.
- 40821 [ECONNRESET] A connection was forcibly closed by a peer.
- 40822 [EINTR] A signal interrupted *sendto()* before any data was transmitted.
- 40823 [EMSGSIZE] The message is too large to be sent all at once, as the socket requires.
- 40824 [ENOTCONN] The socket is connection-mode but is not connected.
- 40825 [ENOTSOCK] The *socket* argument does not refer to a socket.
- 40826 [EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or
40827 more of the values set in *flags*.
- 40828 [EPIPE] The socket is shut down for writing, or the socket is connection-mode and is
40829 no longer connected. In the latter case, and if the socket is of type
40830 SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.
- 40831 If the address family of the socket is AF_UNIX, then *sendto()* shall fail if:
- 40832 [EIO] An I/O error occurred while reading from or writing to the file system.
- 40833 [ELOOP] A loop exists in symbolic links encountered during resolution of the pathname
40834 in the socket address.
- 40835 [ENAMETOOLONG]
40836 A component of a pathname exceeded {NAME_MAX} characters, or an entire
40837 pathname exceeded {PATH_MAX} characters.
- 40838 [ENOENT] A component of the pathname does not name an existing file or the pathname
40839 is an empty string.
- 40840 [ENOTDIR] A component of the path prefix of the pathname in the socket address is not a
40841 directory.
- 40842 The *sendto()* function may fail if:
- 40843 [EACCES] Search permission is denied for a component of the path prefix; or write
40844 access to the named socket is denied.
- 40845 [EDESTADDRREQ]
40846 The socket is not connection-mode and does not have its peer address set, and
40847 no destination address was specified.
- 40848 [EHOSTUNREACH]
40849 The destination host cannot be reached (probably because the host is down or
40850 a remote router cannot reach it).
- 40851 [EINVAL] The *dest_len* argument is not a valid length for the address family.
- 40852 [EIO] An I/O error occurred while reading from or writing to the file system.

40853 [EISCONN] A destination address was specified and the socket is already connected. This
40854 error may or may not be returned for connection mode sockets.

40855 [ENETDOWN] The local network interface used to reach the destination is down.

40856 [ENETUNREACH] No route to the network is present.

40858 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

40859 [ENOMEM] Insufficient memory was available to fulfill the request.

40860 If the address family of the socket is AF_UNIX, then *sendto()* may fail if:

40861 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
40862 resolution of the pathname in the socket address.

40863 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
40864 whose length exceeds {PATH_MAX}.

40866 EXAMPLES

40867 None.

40868 APPLICATION USAGE

40869 The *select()* and *poll()* functions can be used to determine when it is possible to send more data.

40870 RATIONALE

40871 None.

40872 FUTURE DIRECTIONS

40873 None.

40874 SEE ALSO

40875 *getsockopt()*, *poll()*, *recv()*, *recvfrom()*, *recvmsg()*, *select()*, *send()*, *sendmsg()*, *setsockopt()*,
40876 *shutdown()*, *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

40877 CHANGE HISTORY

40878 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

40879 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
40880 [ELOOP] error condition is added.

40881 NAME

40882 *setbuf* — assign buffering to a stream

40883 SYNOPSIS

40884 #include <stdio.h>
40885 void setbuf(FILE *restrict *stream*, char *restrict *buf*);

40886 DESCRIPTION

40887 CX 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.

40890 Except that it returns no value, the function call:

40891 *setbuf*(*stream*, *buf*)

40892 shall be equivalent to:

40893 *setvbuf*(*stream*, *buf*, _IOMB, BUFSIZ)

40894 if *buf* is not a null pointer, or to:

40895 *setvbuf*(*stream*, *buf*, _IONBF, BUFSIZ)

40896 if *buf* is a null pointer.

40897 RETURN VALUE

40898 The *setbuf*() function shall not return a value.

40899 ERRORS

40900 No errors are defined.

40901 EXAMPLES

40902 None.

40903 APPLICATION USAGE

40904 A common source of error is allocating buffer space as an “automatic” variable in a code block,
40905 and then failing to close the stream in the same block.

40906 With *setbuf*(), allocating a buffer of BUFSIZ bytes does not necessarily imply that all of BUFSIZ
40907 bytes are used for the buffer area.

40908 RATIONALE

40909 None.

40910 FUTURE DIRECTIONS

40911 None.

40912 SEE ALSO

40913 *open*(), *setvbuf*(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

40914 CHANGE HISTORY

40915 First released in Issue 1. Derived from Issue 1 of the SVID.

40916 Issue 6

40917 The prototype for *setbuf*() is updated for alignment with the ISO/IEC 9899:1999 standard.

40918 NAME

40919 **setcontext** — set current user context

40920 SYNOPSIS

40921 OB XSI **#include <ucontext.h>**

1

40922 **int setcontext(const ucontext_t *ucp);**

40923

40924 DESCRIPTION

40925 Refer to *getcontext()*.

40926 NAME

40927 setegid — set the effective group ID

40928 SYNOPSIS

40929 #include <unistd.h>
40930 int setegid(gid_t *gid*);

40931 DESCRIPTION

40932 If *gid* is equal to the real group ID or the saved set-group-ID, or if the process has appropriate
40933 privileges, *setegid()* shall set the effective group ID of the calling process to *gid*; the real group
40934 ID, saved set-group-ID, and any supplementary group IDs shall remain unchanged.

40935 The *setegid()* function shall not affect the supplementary group list in any way.

40936 RETURN VALUE

40937 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
40938 indicate the error.

40939 ERRORS

40940 The *setegid()* function shall fail if:

40941 [EINVAL] The value of the *gid* argument is invalid and is not supported by the
40942 implementation.

40943 [EPERM] The process does not have appropriate privileges and *gid* does not match the
40944 real group ID or the saved set-group-ID.

40945 EXAMPLES

40946 None.

40947 APPLICATION USAGE

40948 None.

40949 RATIONALE

40950 Refer to the RATIONALE section in *setuid()*.

40951 FUTURE DIRECTIONS

40952 None.

40953 SEE ALSO

40954 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the
40955 Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

40956 CHANGE HISTORY

40957 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

40958 NAME

40959 setenv — add or change environment variable

40960 SYNOPSIS

40961 CX

```
#include <stdlib.h>
```

40962

```
int setenv(const char *envname, const char *envval, int overwrite);
```

40963

40964 DESCRIPTION

40965 The *setenv()* function shall update or add a variable in the environment of the calling process.
40966 The *envname* argument points to a string containing the name of an environment variable to be
40967 added or altered. The environment variable shall be set to the value to which *envval* points. The
40968 function shall fail if *envname* points to a string which contains an '=' character. If the
40969 environment variable named by *envname* already exists and the value of *overwrite* is non-zero,
40970 the function shall return success and the environment shall be updated. If the environment
40971 variable named by *envname* already exists and the value of *overwrite* is zero, the function shall
40972 return success and the environment shall remain unchanged.

40973 If the application modifies *environ* or the pointers to which it points, the behavior of *setenv()* is
40974 undefined. The *setenv()* function shall update the list of pointers to which *environ* points.

40975 The strings described by *envname* and *envval* are copied by this function.

40976 The *setenv()* function need not be reentrant. A function that is not required to be reentrant is not
40977 required to be thread-safe.

40978 RETURN VALUE

40979 Upon successful completion, zero shall be returned. Otherwise, -1 shall be returned, *errno* set to
40980 indicate the error, and the environment shall be unchanged.

40981 ERRORS

40982 The *setenv()* function shall fail if:

40983 [EINVAL] The *name* argument is a null pointer, points to an empty string, or points to a
40984 string containing an '=' character.

40985 [ENOMEM] Insufficient memory was available to add a variable or its value to the
40986 environment.

40987 EXAMPLES

40988 None.

40989 APPLICATION USAGE

40990 See *exec*, for restrictions on changing the environment in multi-threaded applications.

1

40991 RATIONALE

40992 Unanticipated results may occur if *setenv()* changes the external variable *environ*. In particular,
40993 if the optional *envp* argument to *main()* is present, it is not changed, and thus may point to an
40994 obsolete copy of the environment (as may any other copy of *environ*). However, other than the
40995 aforementioned restriction, the developers of IEEE Std 1003.1-2001 intended that the traditional
40996 method of walking through the environment by way of the *environ* pointer must be supported.

40997 It was decided that *setenv()* should be required by this revision because it addresses a piece of
40998 missing functionality, and does not impose a significant burden on the implementor.

40999 There was considerable debate as to whether the System V *putenv()* function or the BSD *setenv()*
41000 function should be required as a mandatory function. The *setenv()* function was chosen because
41001 it permitted the implementation of the *unsetenv()* function to delete environmental variables,
41002 without specifying an additional interface. The *putenv()* function is available as an XSI

41003	extension.
41004	The standard developers considered requiring that <i>setenv()</i> indicate an error when a call to it would result in exceeding {ARG_MAX}. The requirement was rejected since the condition might be temporary, with the application eventually reducing the environment size. The ultimate success or failure depends on the size at the time of a call to <i>exec</i> , which returns an indication of this error condition.
41009	FUTURE DIRECTIONS
41010	None.
41011	SEE ALSO
41012	<i>exec</i> , <i>getenv()</i> , <i>unsetenv()</i> , the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>, 1
41013	<sys/types.h>, <unistd.h>
41014	CHANGE HISTORY
41015	First released in Issue 6. Derived from the IEEE P1003.1a draft standard.
41016	IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/55 is applied, adding references to <i>exec</i> in 1
41017	the APPLICATION USAGE and SEE ALSO sections. 1

41018 NAME

41019 seteuid — set effective user ID

41020 SYNOPSIS

```
41021        #include <unistd.h>
41022        int seteuid(uid_t uid);
```

41023 DESCRIPTION

41024 If *uid* is equal to the real user ID or the saved set-user-ID, or if the process has appropriate
41025 privileges, *seteuid()* shall set the effective user ID of the calling process to *uid*; the real user ID
41026 and saved set-user-ID shall remain unchanged.

41027 The *seteuid()* function shall not affect the supplementary group list in any way.

41028 RETURN VALUE

41029 Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and *errno* set to
41030 indicate the error.

41031 ERRORS

41032 The *seteuid()* function shall fail if:

41033 [EINVAL] The value of the *uid* argument is invalid and is not supported by the
41034 implementation.

41035 [EPERM] The process does not have appropriate privileges and *uid* does not match the 2
41036 real user ID or the saved set-user-ID. 2

41037 EXAMPLES

41038 None.

41039 APPLICATION USAGE

41040 None.

41041 RATIONALE

41042 Refer to the RATIONALE section in *setuid()*.

41043 FUTURE DIRECTIONS

41044 None.

41045 SEE ALSO

41046 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *setgid()*, *setregid()*, *setreuid()*, *setuid()*, the
41047 Base Definitions volume of IEEE Std 1003.1-2001, *<sys/types.h>*, *<unistd.h>*

41048 CHANGE HISTORY

41049 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

41050 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/123 is applied, making an editorial 2
41051 correction to the [EPERM] error in the ERRORS section. 2

41052 NAME

41053 setgid — set-group-ID

41054 SYNOPSIS

41055 #include <unistd.h>

41056 int setgid(gid_t *gid*);

41057 DESCRIPTION

41058 If the process has appropriate privileges, *setgid()* shall set the real group ID, effective group ID, and the saved set-group-ID of the calling process to *gid*.

41060 If the process does not have appropriate privileges, but *gid* is equal to the real group ID or the
41061 saved set-group-ID, *setgid()* shall set the effective group ID to *gid*; the real group ID and saved
41062 set-group-ID shall remain unchanged.

41063 The *setgid()* function shall not affect the supplementary group list in any way.

41064 Any supplementary group IDs of the calling process shall remain unchanged.

41065 RETURN VALUE

41066 Upon successful completion, 0 is returned. Otherwise, -1 shall be returned and *errno* set to
41067 indicate the error.

41068 ERRORS

41069 The *setgid()* function shall fail if:

41070 [EINVAL] The value of the *gid* argument is invalid and is not supported by the
41071 implementation.

41072 [EPERM] The process does not have appropriate privileges and *gid* does not match the
41073 real group ID or the saved set-group-ID.

41074 EXAMPLES

41075 None.

41076 APPLICATION USAGE

41077 None.

41078 RATIONALE

41079 Refer to the RATIONALE section in *setuid()*.

41080 FUTURE DIRECTIONS

41081 None.

41082 SEE ALSO

41083 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setregid()*, *setreuid()*, *setuid()*, the
41084 Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

41085 CHANGE HISTORY

41086 First released in Issue 1. Derived from Issue 1 of the SVID.

41087 Issue 6

41088 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

41089 The following new requirements on POSIX implementations derive from alignment with the
41090 Single UNIX Specification:

- 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.

- 41094 • Functionality associated with _POSIX_SAVED_IDS is now mandated. This is a FIPS
41095 requirement.
- 41096 The following changes were made to align with the IEEE P1003.1a draft standard:
41097 • The effects of *setgid()* in processes without appropriate privileges are changed.
41098 • A requirement that the supplementary group list is not affected is added.

41099 NAME

41100 setgrent — reset the group database to the first entry

41101 SYNOPSIS

41102 XSI #include <grp.h>

41103 void setgrent(void);

41104

41105 DESCRIPTION

41106 Refer to *endgrent()*.

41107 NAME

41108 **sethostent** — network host database functions

41109 SYNOPSIS

41110 #include <netdb.h>

41111 void sethostent(int *stayopen*);

41112 DESCRIPTION

41113 Refer to *endhostent()*.

41114 NAME

41115 setitimer — set the value of an interval timer

41116 SYNOPSIS

41117 XSI #include <sys/time.h>

```
41118     int setitimer(int which, const struct itimerval *restrict value,  
41119                 struct itimerval *restrict ovalue);  
41120
```

41121 DESCRIPTION

41122 Refer to *getitimer()*.

41123 NAME

41124 `setjmp` — set jump point for a non-local goto

41125 SYNOPSIS

```
41126        #include <setjmp.h>
41127        int setjmp( jmp_buf env );
```

41128 DESCRIPTION

41129 CX The functionality described on this reference page is aligned with the ISO C standard. Any
41130 conflict between the requirements described here and the ISO C standard is unintentional. This
41131 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

41132 A call to `setjmp()` shall save the calling environment in its `env` argument for later use by
41133 `longjmp()`.

41134 It is unspecified whether `setjmp()` is a macro or a function. If a macro definition is suppressed in
41135 order to access an actual function, or a program defines an external identifier with the name
41136 `setjmp`, the behavior is undefined.

41137 An application shall ensure that an invocation of `setjmp()` appears in one of the following
41138 contexts only:

- 41139 • The entire controlling expression of a selection or iteration statement
- 41140 • One operand of a relational or equality operator with the other operand an integral constant
41141 expression, with the resulting expression being the entire controlling expression of a
41142 selection or iteration statement
- 41143 • The operand of a unary ‘!’ operator with the resulting expression being the entire
41144 controlling expression of a selection or iteration
- 41145 • The entire expression of an expression statement (possibly cast to **void**)

41146 If the invocation appears in any other context, the behavior is undefined.

41147 RETURN VALUE

41148 If the return is from a direct invocation, `setjmp()` shall return 0. If the return is from a call to
41149 `longjmp()`, `setjmp()` shall return a non-zero value.

41150 ERRORS

41151 No errors are defined.

41152 EXAMPLES

41153 None.

41154 APPLICATION USAGE

41155 In general, `sigsetjmp()` is more useful in dealing with errors and interrupts encountered in a low-
41156 level subroutine of a program.

41157 RATIONALE

41158 None.

41159 FUTURE DIRECTIONS

41160 None.

41161 SEE ALSO

41162 `longjmp()`, `sigsetjmp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<setjmp.h>`

41163 CHANGE HISTORY

41164 First released in Issue 1. Derived from Issue 1 of the SVID.

41165 Issue 6

41166 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

41167 NAME

41168 setkey — set encoding key (CRYPT)

41169 SYNOPSIS

41170 XSI #include <stdlib.h>

41171 void setkey(const char *key);

41172

41173 DESCRIPTION

41174 The *setkey()* function provides access to an implementation-defined encoding algorithm. The
41175 argument of *setkey()* is an array of length 64 bytes containing only the bytes with numerical
41176 value of 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is
41177 ignored; this gives a 56-bit key which is used by the algorithm. This is the key that shall be used
41178 with the algorithm to encode a string *block* passed to *encrypt()*.

41179 The *setkey()* function shall not change the setting of *errno* if successful. An application wishing to
41180 check for error situations should set *errno* to 0 before calling *setkey()*. If *errno* is non-zero on
41181 return, an error has occurred.

41182 The *setkey()* function need not be reentrant. A function that is not required to be reentrant is not
41183 required to be thread-safe.

41184 RETURN VALUE

41185 No values are returned.

41186 ERRORS

41187 The *setkey()* function shall fail if:

41188 [ENOSYS] The functionality is not supported on this implementation.

41189 EXAMPLES

41190 None.

41191 APPLICATION USAGE

41192 Decoding need not be implemented in all environments. This is related to government
41193 restrictions in some countries on encryption and decryption routines. Historical practice has
41194 been to ship a different version of the encryption library without the decryption feature in the
41195 routines supplied. Thus the exported version of *encrypt()* does encoding but not decoding.

41196 RATIONALE

41197 None.

41198 FUTURE DIRECTIONS

41199 None.

41200 SEE ALSO

41201 *crypt()*, *encrypt()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

41202 CHANGE HISTORY

41203 First released in Issue 1. Derived from Issue 1 of the SVID.

41204 Issue 5

41205 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

41206 NAME

41207 setlocale — set program locale

41208 SYNOPSIS

```
41209     #include <locale.h>
41210     char *setlocale(int category, const char *locale);
```

41211 DESCRIPTION

41212 CX 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.

41215 The *setlocale()* function selects the appropriate piece of the program's locale, as specified by the *category* and *locale* arguments, and may be used to change or query the program's entire locale or portions thereof. The value *LC_ALL* for *category* names the program's entire locale; other values for *category* name only a part of the program's locale:

41219 *LC_COLLATE* Affects the behavior of regular expressions and the collation functions.

41220 *LC_CTYPE* Affects the behavior of regular expressions, character classification, character conversion functions, and wide-character functions.

41222 CX *LC_MESSAGES* Affects what strings are expected by commands and utilities as affirmative or negative responses.

41224 XSI It also affects what strings are given by commands and utilities as affirmative or negative responses, and the content of messages.

41226 *LC_MONETARY* Affects the behavior of functions that handle monetary values.

41227 *LC_NUMERIC* Affects the behavior of functions that handle numeric values.

41228 *LC_TIME* Affects the behavior of the time conversion functions.

41229 The *locale* argument is a pointer to a character string containing the required setting of *category*. The contents of this string are implementation-defined. In addition, the following preset values of *locale* are defined for all settings of *category*:

41232 CX "POSIX" Specifies the minimal environment for C-language translation called the POSIX locale. If *setlocale()* is not invoked, the POSIX locale is the default at entry to *main()*.

41235 "C" Equivalent to "POSIX".

41236 CX " " Specifies an implementation-defined native environment. The determination of the name of the new locale for the specified category depends on the value of the associated environment variables, *LC_** and *LANG*; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale and the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables.

41242 A null pointer Used to direct *setlocale()* to query the current internationalized environment and return the name of the locale.

41244 CX Setting all of the categories of the locale of the process is similar to successively setting each individual category of the locale of the process, except that all error checking is done before any actions are performed. To set all the categories of the locale of the process, *setlocale()* is invoked as:

2
2
2
2
2
2
2
2

41248	<code>setlocale(LC_ALL, " ") ;</code>	2
41249	In this case, <i>setlocale()</i> shall first verify that the values of all the environment variables it needs according to the precedence rules (described in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables) indicate supported locales. If the value of any of these environment variable searches yields a locale that is not supported (and non-null), <i>setlocale()</i> shall return a null pointer and the locale of the process shall not be changed. If all environment variables name supported locales, <i>setlocale()</i> shall proceed as if it had been called for each category, using the appropriate value from the associated environment variable or from the implementation-defined default if there is no such value.	2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

41257	THR The locale state is common to all threads within a process.
-------	--

41258 RETURN VALUE

Upon successful completion, *setlocale()* shall return the string associated with the specified category for the new locale. Otherwise, *setlocale()* shall return a null pointer and the program's locale is not changed.

A null pointer for *locale* causes *setlocale()* to return a pointer to the string associated with the *category* for the program's current locale. The program's locale shall not be changed.

The string returned by *setlocale()* is such that a subsequent call with that string and its associated *category* shall restore that part of the program's locale. The application shall not modify the string returned which may be overwritten by a subsequent call to *setlocale()*.

41267 ERRORS

41268 No errors are defined.

41269 EXAMPLES

41270 None.

41271 APPLICATION USAGE

41272 The following code illustrates how a program can initialize the international environment for one language, while selectively modifying the program's locale such that regular expressions and string operations can be applied to text recorded in a different language:

```
41275      setlocale(LC_ALL, "De") ;
41276      setlocale(LC_COLLATE, "Fr@dict") ;
```

41277 Internationalized programs must call *setlocale()* to initiate a specific language operation. This can be done by calling *setlocale()* as follows:

```
41279      setlocale(LC_ALL, " ") ;
```

41280 Changing the setting of *LC_MESSAGES* has no effect on catalogs that have already been opened by calls to *catopen()*.

41282 RATIONALE

41283 The ISO C standard defines a collection of functions to support internationalization. One of the most significant aspects of these functions is a facility to set and query the *international environment*. The international environment is a repository of information that affects the behavior of certain functionality, namely:

- 41287 1. Character handling
- 41288 2. Collating
- 41289 3. Date/time formatting
- 41290 4. Numeric editing

- 41291 5. Monetary formatting
41292 6. Messaging

41293 The *setlocale()* function provides the application developer with the ability to set all or portions,
41294 called *categories*, of the international environment. These categories correspond to the areas of
41295 functionality mentioned above. The syntax for *setlocale()* is as follows:

41296 `char *setlocale(int category, const char *locale);`

41297 where *category* is the name of one of following categories, namely:

41298 *LC_COLLATE*
41299 *LC_CTYPE*
41300 *LC_MESSAGES*
41301 *LC_MONETARY*
41302 *LC_NUMERIC*
41303 *LC_TIME*

41304 In addition, a special value called *LC_ALL* directs *setlocale()* to set all categories.

41305 There are two primary uses of *setlocale()*:

- 41306 1. Querying the international environment to find out what it is set to
41307 2. Setting the international environment, or *locale*, to a specific value

41308 The behavior of *setlocale()* in these two areas is described below. Since it is difficult to describe
41309 the behavior in words, examples are used to illustrate the behavior of specific uses.

41310 To query the international environment, *setlocale()* is invoked with a specific category and the
41311 NULL pointer as the locale. The NULL pointer is a special directive to *setlocale()* that tells it to
41312 query rather than set the international environment. The following syntax is used to query the
41313 name of the international environment:

41314 `setlocale({LC_ALL, LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, \`
41315 `LC_NUMERIC, LC_TIME}, (char *) NULL);`

41316 The *setlocale()* function shall return the string corresponding to the current international
41317 environment. This value may be used by a subsequent call to *setlocale()* to reset the international
41318 environment to this value. However, it should be noted that the return value from *setlocale()*
41319 may be a pointer to a static area within the function and is not guaranteed to remain unchanged
41320 (that is, it may be modified by a subsequent call to *setlocale()*). Therefore, if the purpose of
41321 calling *setlocale()* is to save the value of the current international environment so it can be
41322 changed and reset later, the return value should be copied to an array of **char** in the calling
41323 program.

41324 There are three ways to set the international environment with *setlocale()*:

41325 *setlocale(category, string)*

41326 This usage sets a specific *category* in the international environment to a specific value
41327 corresponding to the value of the *string*. A specific example is provided below:

41328 `setlocale(LC_ALL, "fr_FR.ISO-8859-1");`

41329 In this example, all categories of the international environment are set to the locale
41330 corresponding to the string "fr_FR.ISO-8859-1", or to the French language as spoken in
41331 France using the ISO/IEC 8859-1:1998 standard codeset.

41332 If the string does not correspond to a valid locale, *setlocale()* shall return a NULL pointer
41333 and the international environment is not changed. Otherwise, *setlocale()* shall return the

41334 name of the locale just set.

41335 *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.

41339 *setlocale(category, "")*
This sets a specific category to an implementation-defined default. This corresponds to the
value of the environment variables.

41342 **FUTURE DIRECTIONS**

41343 None.

41344 **SEE ALSO**

41345 *exec*, *isalnum()*, *isalpha()*, *isblank()*, *iscntrl()*, *isdigit()*, *isgraph()*, *islower()*, *isprint()*, *ispunct()*,
41346 *isspace()*, *isupper()*, *iswalnum()*, *iswalpha()*, *iswblank()*, *iswcntrl()*, *iswctype()*, *iswdigit()*,
41347 *iswgraph()*, *iswlower()*, *iswprint()*, *iswpunct()*, *iswspace()*, *iswupper()*, *iswxdigit()*, *isxdigit()*,
41348 *localeconv()*, *mblen()*, *mbstowcs()*, *mbtowc()*, *nl_langinfo()*, *printf()*, *scanf()*, *setlocale()*, *strcoll()*,
41349 *strerror()*, *strfmon()*, *strtod()*, *strxfrm()*, *tolower()*, *toupper()*, *towlower()*, *towupper()*, *wcscoll()*,
41350 *wctod()*, *wctombs()*, *wcsxfrm()*, *wctomb()*, the Base Definitions volume of IEEE Std 1003.1-2001,
41351 <*langinfo.h*>, <*locale.h*>

41352 **CHANGE HISTORY**

41353 First released in Issue 3.

41354 **Issue 5**

41355 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

41356 **Issue 6**

41357 Extensions beyond the ISO C standard are marked.

41358 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

41359 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/124 is applied, updating the 2
41360 DESCRIPTION to clarify the behavior of: 2

41361 *setlocale(LC_ALL, " ")*; 2

41362 NAME

41363 setlogmask — set the log priority mask

41364 SYNOPSIS

41365 XSI #include <syslog.h>

41366 int setlogmask(int *maskpri*);

41367

41368 DESCRIPTION

41369 Refer to *closelog()*.

41370 **NAME**41371 **setnetent** — network database function41372 **SYNOPSIS**

41373 #include <netdb.h>

41374 void setnetent(int stayopen);

41375 **DESCRIPTION**41376 Refer to *endnetent()*.

41377 NAME

41378 `setpgid` — set process group ID for job control

41379 SYNOPSIS

```
41380        #include <unistd.h>
41381        int setpgid(pid_t pid, pid_t pgid);
```

41382 DESCRIPTION

41383 The `setpgid()` function shall either join an existing process group or create a new process group
41384 within the session of the calling process. The process group ID of a session leader shall not
41385 change. Upon successful completion, the process group ID of the process with a process ID that
41386 matches *pid* shall be set to *pgid*. As a special case, if *pid* is 0, the process ID of the calling process
41387 shall be used. Also, if *pgid* is 0, the process ID of the indicated process shall be used.

1

41388 RETURN VALUE

41389 Upon successful completion, `setpgid()` shall return 0; otherwise, -1 shall be returned and *errno*
41390 shall be set to indicate the error.

41391 ERRORS

41392 The `setpgid()` function shall fail if:

- | | | |
|-------|----------|---|
| 41393 | [EACCES] | The value of the <i>pid</i> argument matches the process ID of a child process of the calling process and the child process has successfully executed one of the <code>exec</code> functions. |
| 41396 | [EINVAL] | The value of the <i>pgid</i> argument is less than 0, or is not a value supported by the implementation. |
| 41398 | [EPERM] | The process indicated by the <i>pid</i> argument is a session leader. |
| 41399 | [EPERM] | The value of the <i>pid</i> argument matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process. |
| 41402 | [EPERM] | The value of the <i>pgid</i> argument is valid but does not match the process ID of the process indicated by the <i>pid</i> argument and there is no process with a process group ID that matches the value of the <i>pgid</i> argument in the same session as the calling process. |
| 41406 | [ESRCH] | The value of the <i>pid</i> argument does not match the process ID of the calling process or of a child process of the calling process. |

41408 EXAMPLES

41409 None.

41410 APPLICATION USAGE

41411 None.

41412 RATIONALE

41413 The `setpgid()` function shall group processes together for the purpose of signaling, placement in foreground or background, and other job control actions.

41415 The `setpgid()` function is similar to the `setpgrp()` function of 4.2 BSD, except that 4.2 BSD allowed
41416 the specified new process group to assume any value. This presents certain security problems
41417 and is more flexible than necessary to support job control.

41418 To provide tighter security, `setpgid()` only allows the calling process to join a process group
41419 already in use inside its session or create a new process group whose process group ID was
41420 equal to its process ID.

41421 When a job control shell spawns a new job, the processes in the job must be placed into a new
41422 process group via *setpgid()*. There are two timing constraints involved in this action:

- 41423 1. The new process must be placed in the new process group before the appropriate program
41424 is launched via one of the *exec* functions.
- 41425 2. The new process must be placed in the new process group before the shell can correctly
41426 send signals to the new process group.

41427 To address these constraints, the following actions are performed. The new processes call
41428 *setpgid()* to alter their own process groups after *fork()* but before *exec*. This satisfies the first
41429 constraint. Under 4.3 BSD, the second constraint is satisfied by the synchronization property of
41430 *vfork()*; that is, the shell is suspended until the child has completed the *exec*, thus ensuring that
41431 the child has completed the *setpgid()*. A new version of *fork()* with this same synchronization
41432 property was considered, but it was decided instead to merely allow the parent shell process to
41433 adjust the process group of its child processes via *setpgid()*. Both timing constraints are now
41434 satisfied by having both the parent shell and the child attempt to adjust the process group of the
41435 child process; it does not matter which succeeds first.

41436 Since it would be confusing to an application to have its process group change after it began
41437 executing (that is, after *exec*), and because the child process would already have adjusted its
41438 process group before this, the [EACCES] error was added to disallow this.

41439 One non-obvious use of *setpgid()* is to allow a job control shell to return itself to its original
41440 process group (the one in effect when the job control shell was executed). A job control shell
41441 does this before returning control back to its parent when it is terminating or suspending itself as
41442 a way of restoring its job control “state” back to what its parent would expect. (Note that the
41443 original process group of the job control shell typically matches the process group of its parent,
41444 but this is not necessarily always the case.)

41445 FUTURE DIRECTIONS

41446 None.

41447 SEE ALSO

41448 *exec*, *getpgrp()*, *setsid()*, *tcsetpgrp()*, the Base Definitions volume of IEEE Std 1003.1-2001,
41449 <sys/types.h>, <unistd.h>

41450 CHANGE HISTORY

41451 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

41452 Issue 6

41453 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

41454 The following new requirements on POSIX implementations derive from alignment with the
41455 Single UNIX Specification:

- 41456 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
41457 required for conforming implementations of previous POSIX specifications, it was not
41458 required for UNIX applications.
- 41459 • The *setpgid()* function is mandatory since _POSIX_JOB_CONTROL is required to be defined
41460 in this issue. This is a FIPS requirement.

41461 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/56 is applied, changing the wording in 1
41462 the DESCRIPTION from “the process group ID of the indicated process shall be used” to “the 1
41463 process ID of the indicated process shall be used”. This change reverts the wording to as in the 1
41464 ISO POSIX-1:1996 standard; it appeared to be an unintentional change. 1

41465 NAME

41466 *setpgrp* — set the process group ID

41467 SYNOPSIS

41468 XSI #include <unistd.h>
41469
41470 pid_t setpgrp(void);

41471 DESCRIPTION

41472 If the calling process is not already a session leader, *setpgrp()* sets the process group ID of the calling process to the process ID of the calling process. If *setpgrp()* creates a new session, then the new session has no controlling terminal.

41475 The *setpgrp()* function has no effect when the calling process is a session leader.

41476 RETURN VALUE

41477 Upon completion, *setpgrp()* shall return the process group ID.

41478 ERRORS

41479 No errors are defined.

41480 EXAMPLES

41481 None.

41482 APPLICATION USAGE

41483 None.

41484 RATIONALE

41485 None.

41486 FUTURE DIRECTIONS

41487 None.

41488 SEE ALSO

41489 *exec*, *fork()*, *getpid()*, *getsid()*, *kill()*, *setpgid()*, *setsid()*, the Base Definitions volume of
41490 IEEE Std 1003.1-2001, <unistd.h>

41491 CHANGE HISTORY

41492 First released in Issue 4, Version 2.

41493 Issue 5

41494 Moved from X/OPEN UNIX extension to BASE.

41495 NAME

41496 setpriority — set the nice value

41497 SYNOPSIS

41498 XSI #include <sys/resource.h>

41499 int setpriority(int which, id_t who, int nice);

41500

41501 DESCRIPTION

41502 Refer to *getpriority()*.

41503 **NAME**
41504 `setprotoent` — network protocol database functions

41505 **SYNOPSIS**
41506 `#include <netdb.h>`
41507 `void setprotoent(int stayopen);`

41508 **DESCRIPTION**
41509 Refer to *endprotoent()*.

41510 NAME

41511 `setpwent` — user database function

41512 SYNOPSIS

41513 XSI `#include <pwd.h>`

41514 `void setpwent(void);`

41515

41516 DESCRIPTION

41517 Refer to *endpwent()*.

41518 NAME

41519 setregid — set real and effective group IDs

41520 SYNOPSIS

41521 XSI #include <unistd.h>

41522 int setregid(gid_t rgid, gid_t egid);

41523

41524 DESCRIPTION

41525 The *setregid()* function shall set the real and effective group IDs of the calling process.

41526 If *rgid* is -1, the real group ID shall not be changed; if *egid* is -1, the effective group ID shall not be changed.

41528 The real and effective group IDs may be set to different values in the same call.

41529 Only a process with appropriate privileges can set the real group ID and the effective group ID to any valid value.

41531 A non-privileged process can set either the real group ID to the saved set-group-ID from one of
41532 the *exec* family of functions, or the effective group ID to the saved set-group-ID or the real group
41533 ID.

41534 Any supplementary group IDs of the calling process remain unchanged.

41535 RETURN VALUE

41536 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
41537 indicate the error, and neither of the group IDs are changed.

41538 ERRORS

41539 The *setregid()* function shall fail if:

41540 [EINVAL] The value of the *rgid* or *egid* argument is invalid or out-of-range.

41541 [EPERM] The process does not have appropriate privileges and a change other than
41542 changing the real group ID to the saved set-group-ID, or changing the
41543 effective group ID to the real group ID or the saved set-group-ID, was
41544 requested.

41545 EXAMPLES

41546 None.

41547 APPLICATION USAGE

41548 If a set-group-ID process sets its effective group ID to its real group ID, it can still set its effective
41549 group ID back to the saved set-group-ID.

41550 RATIONALE

41551 None.

41552 FUTURE DIRECTIONS

41553 None.

41554 SEE ALSO

41555 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setreuid()*, *setuid()*, the
41556 Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

41557 CHANGE HISTORY

41558 First released in Issue 4, Version 2.

41559 Issue 5

41560 Moved from X/OPEN UNIX extension to BASE.

41561 The DESCRIPTION is updated to indicate that the saved set-group-ID can be set by any of the exec family of functions, not just *execve()*.
41562

41563 NAME

41564 setreuid — set real and effective user IDs

41565 SYNOPSIS

41566 XSI #include <unistd.h>

41567 int setreuid(uid_t ruid, uid_t euid);

41568

41569 DESCRIPTION

41570 The *setreuid()* function shall set the real and effective user IDs of the current process to the values specified by the *ruid* and *euid* arguments. If *ruid* or *euid* is -1, the corresponding effective or real user ID of the current process shall be left unchanged.

41573 A process with appropriate privileges can set either ID to any value. An unprivileged process can only set the effective user ID if the *euid* argument is equal to either the real, effective, or saved user ID of the process.

41576 It is unspecified whether a process without appropriate privileges is permitted to change the real user ID to match the current real, effective, or saved set-user-ID of the process.

41578 RETURN VALUE

41579 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

41581 ERRORS

41582 The *setreuid()* function shall fail if:

41583 [EINVAL] The value of the *ruid* or *euid* argument is invalid or out-of-range.

41584 [EPERM] The current process does not have appropriate privileges, and either an attempt was made to change the effective user ID to a value other than the real user ID or the saved set-user-ID or an attempt was made to change the real user ID to a value not permitted by the implementation.

41588 EXAMPLES**41589 Setting the Effective User ID to the Real User ID**

41590 The following example sets the effective user ID of the calling process to the real user ID, so that files created later will be owned by the current user.

```
41592        #include <unistd.h>
41593        #include <sys/types.h>
41594        ...
41595        setreuid(getuid(), getuid());
41596        ...
```

41597 APPLICATION USAGE

41598 None.

41599 RATIONALE

41600 None.

41601 FUTURE DIRECTIONS

41602 None.

41603 SEE ALSO

41604 *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setuid()*, the Base
41605 Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

41606 CHANGE HISTORY

41607 First released in Issue 4, Version 2.

41608 Issue 5

41609 Moved from X/OPEN UNIX extension to BASE.

41610 NAME

41611 **setrlimit** — control maximum resource consumption

41612 SYNOPSIS

41613 XSI #include <sys/resource.h>

41614 int setrlimit(int *resource*, const struct rlimit **rlp*);

41615

41616 DESCRIPTION

41617 Refer to *getrlimit()*.

41618 **NAME**41619 **setservent** — network services database functions41620 **SYNOPSIS**

41621 #include <netdb.h>

41622 void setservent(int *stayopen*);41623 **DESCRIPTION**41624 Refer to *endservent*().

41625 NAME

41626 setsid — create session and set process group ID

41627 SYNOPSIS

```
41628       #include <unistd.h>
41629       pid_t setsid(void);
```

41630 DESCRIPTION

41631 The *setsid()* function shall create a new session, if the calling process is not a process group leader. Upon return the calling process shall be the session leader of this new session, shall be the process group leader of a new process group, and shall have no controlling terminal. The process group ID of the calling process shall be set equal to the process ID of the calling process. The calling process shall be the only process in the new process group and the only process in the new session.

41637 RETURN VALUE

41638 Upon successful completion, *setsid()* shall return the value of the new process group ID of the calling process. Otherwise, it shall return (**pid_t**)–1 and set *errno* to indicate the error.

41640 ERRORS

41641 The *setsid()* function shall fail if:

41642 [EPERM] The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.

41645 EXAMPLES

41646 None.

41647 APPLICATION USAGE

41648 None.

41649 RATIONALE

41650 The *setsid()* function is similar to the *setpgrp()* function of System V. System V, without job control, groups processes into process groups and creates new process groups via *setpgrp()*; only one process group may be part of a login session.

41653 Job control allows multiple process groups within a login session. In order to limit job control actions so that they can only affect processes in the same login session, this volume of IEEE Std 1003.1-2001 adds the concept of a session that is created via *setsid()*. The *setsid()* function also creates the initial process group contained in the session. Additional process groups can be created via the *setpgid()* function. A System V process group would correspond to a POSIX System Interfaces session containing a single POSIX process group. Note that this function requires that the calling process not be a process group leader. The usual way to ensure this is true is to create a new process with *fork()* and have it call *setsid()*. The *fork()* function guarantees that the process ID of the new process does not match any existing process group ID.

41662 FUTURE DIRECTIONS

41663 None.

41664 SEE ALSO

41665 *getsid()*, *setpgid()*, *setpgrp()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<sys/types.h>**,
41666 **<unistd.h>**

41667 CHANGE HISTORY

41668 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

41669 Issue 6

41670 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

41671 The following new requirements on POSIX implementations derive from alignment with the
41672 Single UNIX Specification:

- 41673 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
41674 required for conforming implementations of previous POSIX specifications, it was not
41675 required for UNIX applications.

41676 NAME

41677 setsockopt — set the socket options

41678 SYNOPSIS

```
41679 #include <sys/socket.h>
41680 int setsockopt(int socket, int level, int option_name,
41681     const void *option_value, socklen_t option_len);
```

41682 DESCRIPTION

41683 The *setsockopt()* function shall set the option specified by the *option_name* argument, at the
 41684 protocol level specified by the *level* argument, to the value pointed to by the *option_value*
 41685 argument for the socket associated with the file descriptor specified by the *socket* argument.

41686 The *level* argument specifies the protocol level at which the option resides. To set options at the
 41687 socket level, specify the *level* argument as SOL_SOCKET. To set options at other levels, supply
 41688 the appropriate *level* identifier for the protocol controlling the option. For example, to indicate
 41689 that an option is interpreted by the TCP (Transport Control Protocol), set *level* to IPPROTO_TCP
 41690 as defined in the <netinet/in.h> header.

41691 The *option_name* argument specifies a single option to set. The *option_name* argument and any
 41692 specified options are passed uninterpreted to the appropriate protocol module for
 41693 interpretations. The <sys/socket.h> header defines the socket-level options. The options are as
 41694 follows:

41695 SO_DEBUG	Turns on recording of debugging information. This option enables or 41696 disables debugging in the underlying protocol modules. This option takes 41697 an int value. This is a Boolean option.
41698 SO_BROADCAST	Permits sending of broadcast messages, if this is supported by the 41699 protocol. This option takes an int value. This is a Boolean option.
41700 SO_REUSEADDR	Specifies that the rules used in validating addresses supplied to <i>bind()</i> 41701 should allow reuse of local addresses, if this is supported by the protocol. 41702 This option takes an int value. This is a Boolean option.
41703 SO_KEEPALIVE	Keeps connections active by enabling the periodic transmission of 41704 messages, if this is supported by the protocol. This option takes an int 41705 value. 41706 If the connected socket fails to respond to these messages, the connection 41707 is broken and threads writing to that socket are notified with a SIGPIPE 41708 signal. This is a Boolean option.
41709 SO_LINGER	Lingers on a <i>close()</i> if data is present. This option controls the action 41710 taken when unsent messages queue on a socket and <i>close()</i> is performed. 41711 If SO_LINGER is set, the system shall block the calling thread during 41712 <i>close()</i> until it can transmit the data or until the time expires. If 41713 SO_LINGER is not specified, and <i>close()</i> is issued, the system handles the 41714 call in a way that allows the calling thread to continue as quickly as 41715 possible. This option takes a linger structure, as defined in the 41716 <sys/socket.h> header, to specify the state of the option and linger 41717 interval.
41718 SO_OOBINLINE	Leaves received out-of-band data (data marked urgent) inline. This 41719 option takes an int value. This is a Boolean option.
41720 SO_SNDBUF	Sets send buffer size. This option takes an int value.

2

2

41721	SO_RCVBUF	Sets receive buffer size. This option takes an int value.
41722	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.
41723		
41724		
41725		
41726		
41727	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.
41728		
41729		
41730		
41731		
41732		
41733		
41734		
41735	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.
41736		
41737		
41738		
41739		
41740		
41741		
41742		
41743		
41744		
41745	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.
41746		
41747		
41748		
41749		
41750	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.
41751		
41752		
41753		
41754		
41755		
41756		
41757	For Boolean options, 0 indicates that the option is disabled and 1 indicates that the option is enabled.	
41758		
41759	Options at other protocol levels vary in format and name.	
41760	RETURN VALUE	
41761	Upon successful completion, <i>setsockopt()</i> shall return 0. Otherwise, -1 shall be returned and <i>errno</i> set to indicate the error.	
41762		
41763	ERRORS	
41764	The <i>setsockopt()</i> function shall fail if:	
41765	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
41766	[EDOM]	The send and receive timeout values are too big to fit into the timeout fields in the socket structure.
41767		

41768	[EINVAL]	The specified option is invalid at the specified socket level or the socket has been shut down.
41769		
41770	[EISCONN]	The socket is already connected, and a specified option cannot be set while the socket is connected.
41771		
41772	[ENOPROTOOPT]	
41773		The option is not supported by the protocol.
41774	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
41775		The <i>setsockopt()</i> function may fail if:
41776	[ENOMEM]	There was insufficient memory available for the operation to complete.
41777	[ENOBUFS]	Insufficient resources are available in the system to complete the call.

41778 EXAMPLES

41779 None.

41780 APPLICATION USAGE

41781 The *setsockopt()* function provides an application program with the means to control socket behavior. An application program can use *setsockopt()* to allocate buffer space, control timeouts, or permit socket data broadcasts. The <sys/socket.h> header defines the socket-level options available to *setsockopt()*.

41785 Options may exist at multiple protocol levels. The SO_ options are always present at the uppermost socket level.

41787 RATIONALE

41788 None.

41789 FUTURE DIRECTIONS

41790 None.

41791 SEE ALSO

41792 Section 2.10 (on page 59), *bind()*, *endprotoent()*, *getsockopt()*, *socket()*, the Base Definitions volume
41793 of IEEE Std 1003.1-2001, <netinet/in.h>, <sys/socket.h>

41794 CHANGE HISTORY

41795 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

41796 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/125 is applied, updating the SOLINGER 2
41797 option in the DESCRIPTION to refer to the calling thread rather than the process. 2

41798 NAME

41799 **setstate** — switch pseudo-random number generator state arrays

41800 SYNOPSIS

41801 XSI `#include <stdlib.h>`

41802 `char *setstate(const char *state);`

41803

41804 DESCRIPTION

41805 Refer to *initstate()*.

41806 NAME

41807 **setuid** — set user ID

41808 SYNOPSIS

```
41809        #include <unistd.h>
41810        int setuid(uid_t uid);
```

41811 DESCRIPTION

41812 If the process has appropriate privileges, *setuid()* shall set the real user ID, effective user ID, and the saved set-user-ID of the calling process to *uid*.

41814 If the process does not have appropriate privileges, but *uid* is equal to the real user ID or the saved set-user-ID, *setuid()* shall set the effective user ID to *uid*; the real user ID and saved set-user-ID shall remain unchanged.

41817 The *setuid()* function shall not affect the supplementary group list in any way.

41818 RETURN VALUE

41819 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

41821 ERRORS

41822 The *setuid()* function shall fail, return -1, and set *errno* to the corresponding value if one or more of the following are true:

41824 [EINVAL] The value of the *uid* argument is invalid and not supported by the implementation.

41826 [EPERM] The process does not have appropriate privileges and *uid* does not match the real user ID or the saved set-user-ID.

41828 EXAMPLES

41829 None.

41830 APPLICATION USAGE

41831 None.

41832 RATIONALE

41833 The various behaviors of the *setuid()* and *setgid()* functions when called by non-privileged processes reflect the behavior of different historical implementations. For portability, it is recommended that new non-privileged applications use the *seteuid()* and *setegid()* functions instead.

41837 The saved set-user-ID capability allows a program to regain the effective user ID established at the last *exec* call. Similarly, the saved set-group-ID capability allows a program to regain the effective group ID established at the last *exec* call. These capabilities are derived from System V. Without them, a program might have to run as superuser in order to perform the same functions, because superuser can write on the user's files. This is a problem because such a program can write on any user's files, and so must be carefully written to emulate the permissions of the calling process properly. In System V, these capabilities have traditionally been implemented only via the *setuid()* and *setgid()* functions for non-privileged processes. The fact that the behavior of those functions was different for privileged processes made them difficult to use. The POSIX.1-1990 standard defined the *setuid()* function to behave differently for privileged and unprivileged users. When the caller had the appropriate privilege, the function set the calling process' real user ID, effective user ID, and saved set-user ID on implementations that supported it. When the caller did not have the appropriate privilege, the function set only the effective user ID, subject to permission checks. The former use is generally needed for utilities like *login* and *su*, which are not conforming applications and thus outside the

41852 scope of IEEE Std 1003.1-2001. These utilities wish to change the user ID irrevocably to a new
41853 value, generally that of an unprivileged user. The latter use is needed for conforming
41854 applications that are installed with the set-user-ID bit and need to perform operations using the
41855 real user ID.

41856 IEEE Std 1003.1-2001 augments the latter functionality with a mandatory feature named
41857 `_POSIX_SAVED_IDS`. This feature permits a set-user-ID application to switch its effective user
41858 ID back and forth between the values of its exec-time real user ID and effective user ID.
41859 Unfortunately, the POSIX.1-1990 standard did not permit a conforming application using this
41860 feature to work properly when it happened to be executed with the (implementation-defined)
41861 appropriate privilege. Furthermore, the application did not even have a means to tell whether it
41862 had this privilege. Since the saved set-user-ID feature is quite desirable for applications, as
41863 evidenced by the fact that NIST required it in FIPS 151-2, it has been mandated by
41864 IEEE Std 1003.1-2001. However, there are implementors who have been reluctant to support it
41865 given the limitation described above.

41866 The 4.3BSD system handles the problem by supporting separate functions: `setuid()` (which
41867 always sets both the real and effective user IDs, like `setuid()` in IEEE Std 1003.1-2001 for
41868 privileged users), and `seteuid()` (which always sets just the effective user ID, like `setuid()` in
41869 IEEE Std 1003.1-2001 for non-privileged users). This separation of functionality into distinct
41870 functions seems desirable. 4.3BSD does not support the saved set-user-ID feature. It supports
41871 similar functionality of switching the effective user ID back and forth via `setreuid()`, which
41872 permits reversing the real and effective user IDs. This model seems less desirable than the saved
41873 set-user-ID because the real user ID changes as a side effect. The current 4.4BSD includes saved
41874 effective IDs and uses them for `seteuid()` and `setegid()` as described above. The `setreuid()` and
41875 `setregid()` functions will be deprecated or removed.

41876 The solution here is:

- 41877 • Require that all implementations support the functionality of the saved set-user-ID, which is
41878 set by the `exec` functions and by privileged calls to `setuid()`.
- 41879 • Add the `seteuid()` and `setegid()` functions as portable alternatives to `setuid()` and `setgid()` for
41880 non-privileged and privileged processes.

41881 Historical systems have provided two mechanisms for a set-user-ID process to change its
41882 effective user ID to be the same as its real user ID in such a way that it could return to the
41883 original effective user ID: the use of the `setuid()` function in the presence of a saved set-user-ID,
41884 or the use of the BSD `setreuid()` function, which was able to swap the real and effective user IDs.
41885 The changes included in IEEE Std 1003.1-2001 provide a new mechanism using `seteuid()` in
41886 conjunction with a saved set-user-ID. Thus, all implementations with the new `seteuid()`
41887 mechanism will have a saved set-user-ID for each process, and most of the behavior controlled
41888 by `_POSIX_SAVED_IDS` has been changed to agree with the case where the option was defined.
41889 The `kill()` function is an exception. Implementors of the new `seteuid()` mechanism will generally
41890 be required to maintain compatibility with the older mechanisms previously supported by their
41891 systems. However, compatibility with this use of `setreuid()` and with the `_POSIX_SAVED_IDS`
41892 behavior of `kill()` is unfortunately complicated. If an implementation with a saved set-user-ID
41893 allows a process to use `setreuid()` to swap its real and effective user IDs, but were to leave the
41894 saved set-user-ID unmodified, the process would then have an effective user ID equal to the
41895 original real user ID, and both real and saved set-user-ID would be equal to the original effective
41896 user ID. In that state, the real user would be unable to kill the process, even though the effective
41897 user ID of the process matches that of the real user, if the `kill()` behavior of `_POSIX_SAVED_IDS`
41898 was used. This is obviously not acceptable. The alternative choice, which is used in at least one
41899 implementation, is to change the saved set-user-ID to the effective user ID during most calls to
41900 `setreuid()`. The standard developers considered that alternative to be less correct than the

41901 retention of the old behavior of *kill()* in such systems. Current conforming applications shall
41902 accommodate either behavior from *kill()*, and there appears to be no strong reason for *kill()* to
41903 check the saved set-user-ID rather than the effective user ID.

41904 **FUTURE DIRECTIONS**

41905 None.

41906 **SEE ALSO**

41907 *exec*, *getegid()*, *geteuid()*, *getgid()*, *getuid()*, *setegid()*, *seteuid()*, *setgid()*, *setregid()*, *setreuid()*, the
41908 Base Definitions volume of IEEE Std 1003.1-2001, <**sys/types.h**>, <**unistd.h**>

41909 **CHANGE HISTORY**

41910 First released in Issue 1. Derived from Issue 1 of the SVID.

41911 **Issue 6**

41912 In the SYNOPSIS, the optional include of the <**sys/types.h**> header is removed.

41913 The following new requirements on POSIX implementations derive from alignment with the
41914 Single UNIX Specification:

- 41915 • The requirement to include <**sys/types.h**> has been removed. Although <**sys/types.h**> was
41916 required for conforming implementations of previous POSIX specifications, it was not
41917 required for UNIX applications.
- 41918 • The functionality associated with _POSIX_SAVED_IDS is now mandatory. This is a FIPS
41919 requirement.

41920 The following changes were made to align with the IEEE P1003.1a draft standard:

- 41921 • The effects of *setuid()* in processes without appropriate privileges are changed.
- 41922 • A requirement that the supplementary group list is not affected is added.

41923 **NAME**41924 **setutxent** — reset the user accounting database to the first entry41925 **SYNOPSIS**41926 XSI

```
#include <utmpx.h>
```

41927

```
void setutxent(void);
```

41928

41929 **DESCRIPTION**41930 Refer to *endutxent()*.

41931 NAME

41932 setvbuf — assign buffering to a stream

41933 SYNOPSIS

```
41934     #include <stdio.h>
41935     int setvbuf(FILE *restrict stream, char *restrict buf, int type,
41936           size_t size);
```

41937 DESCRIPTION

41938 CX 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.

41941 The *setvbuf()* function may be used after the stream pointed to by *stream* is associated with an open file but before any other operation (other than an unsuccessful call to *setvbuf()*) is performed on the stream. The argument *type* determines how *stream* shall be buffered, as follows:

- *{_IOFBF}* shall cause input/output to be fully buffered.
- *{_IOLBF}* shall cause input/output to be line buffered.
- *{_IONBF}* shall cause input/output to be unbuffered.

41948 If *buf* is not a null pointer, the array it points to may be used instead of a buffer allocated by *setvbuf()* and the argument *size* specifies the size of the array; otherwise, *size* may determine the size of a buffer allocated by the *setvbuf()* function. The contents of the array at any time are unspecified.

41952 For information about streams, see Section 2.5 (on page 34).

41953 RETURN VALUE

41954 Upon successful completion, *setvbuf()* shall return 0. Otherwise, it shall return a non-zero value

41949 CX if an invalid value is given for *type* or if the request cannot be honored, and may set *errno* to

41955 indicate the error.

41957 ERRORS

41958 The *setvbuf()* function may fail if:

41959 CX [EBADF] The file descriptor underlying *stream* is not valid.

41960 EXAMPLES

41961 None.

41962 APPLICATION USAGE

41963 A common source of error is allocating buffer space as an “automatic” variable in a code block,

41964 and then failing to close the stream in the same block.

41965 With *setvbuf()*, allocating a buffer of *size* bytes does not necessarily imply that all of *size* bytes are

41966 used for the buffer area.

41967 Applications should note that many implementations only provide line buffering on input from

41968 terminal devices.

41969 RATIONALE

41970 None.

41971 FUTURE DIRECTIONS

41972 None.

41973 SEE ALSO

41974 Section 2.5 (on page 34), *fopen()*, *setbuf()*, the Base Definitions volume of IEEE Std 1003.1-2001,
41975 <**stdio.h**>

41976 CHANGE HISTORY

41977 First released in Issue 1. Derived from Issue 1 of the SVID.

41978 Issue 6

41979 Extensions beyond the ISO C standard are marked.

41980 The *setvbuf()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

41981 NAME

41982 shm_open — open a shared memory object (REALTIME)

41983 SYNOPSIS

41984 SHM #include <sys/mman.h>

41985 int shm_open(const char *name, int oflag, mode_t mode);

41986

41987 DESCRIPTION

41988 The *shm_open()* function shall establish a connection between a shared memory object and a file descriptor. It shall create an open file description that refers to the shared memory object and a file descriptor that refers to that open file description. The file descriptor is used by other functions to refer to that shared memory object. The *name* argument points to a string naming a shared memory object. It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments. The *name* argument conforms to the construction rules for a pathname. If *name* begins with the slash character, then processes calling *shm_open()* with the same value of *name* refer to the same shared memory object, as long as that name has not been removed. If *name* does not begin with the slash character, the effect is implementation-defined. The interpretation of slash characters other than the leading slash character in *name* is implementation-defined.

41999 If successful, *shm_open()* shall return a file descriptor for the shared 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 does 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 is set.

42004 The file status flags and file access modes of the open file description are according to the value of *oflag*. The *oflag* argument is the bitwise-inclusive OR of the following flags defined in the <fcntl.h> header. Applications specify exactly one of the first two values (access modes) below in the value of *oflag*:

42008 O_RDONLY Open for read access only.

42009 O_RDWR Open for read or write access.

42010 Any combination of the remaining flags may be specified in the value of *oflag*:

42011 O_CREAT If the shared memory object exists, this flag has no effect, except as noted under O_EXCL below. Otherwise, the shared memory object is created; the user ID of the shared memory object shall be set to the effective user ID of the process; the group ID of the shared memory object is set to a system default group ID or to the effective group ID of the process. The permission bits of the shared memory object shall be set to the value of the *mode* argument except those set in the file mode creation mask of the process. When bits in *mode* other than the file permission bits are set, the effect is unspecified. The *mode* argument does not affect whether the shared memory object is opened for reading, for writing, or for both. The shared memory object has a size of zero.

42021 O_EXCL If O_EXCL and O_CREAT are set, *shm_open()* fails if the shared memory object exists. The check for the existence of the shared memory object and the creation of the object if it does not exist is atomic with respect to other processes executing *shm_open()* naming the same shared memory object with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the result is undefined.

42027 O_TRUNC If the shared memory object exists, and it is successfully opened O_RDWR, 42028 the object shall be truncated to zero length and the mode and owner shall be 42029 unchanged by this function call. The result of using O_TRUNC with 42030 O_RDONLY is undefined.

42031 When a shared memory object is created, the state of the shared memory object, including all 42032 data associated with the shared memory object, persists until the shared memory object is 42033 unlinked and all other references are gone. It is unspecified whether the name and shared 42034 memory object state remain valid after a system reboot.

42035 RETURN VALUE

42036 Upon successful completion, the *shm_open()* function shall return a non-negative integer 42037 representing the lowest numbered unused file descriptor. Otherwise, it shall return -1 and set 42038 *errno* to indicate the error.

42039 ERRORS

42040 The *shm_open()* function shall fail if:

42041 [EACCES] The shared memory object exists and the permissions specified by *oflag* are 42042 denied, or the shared memory object does not exist and permission to create 42043 the shared memory object is denied, or O_TRUNC is specified and write 42044 permission is denied.

42045 [EEXIST] O_CREAT and O_EXCL are set and the named shared memory object already 42046 exists.

42047 [EINTR] The *shm_open()* operation was interrupted by a signal.

42048 [EINVAL] The *shm_open()* operation is not supported for the given name.

42049 [EMFILE] Too many file descriptors are currently in use by this process.

42050 [ENAMETOOLONG] The length of the *name* argument exceeds {PATH_MAX} or a pathname 42051 component is longer than {NAME_MAX}.

42053 [ENFILE] Too many shared memory objects are currently open in the system.

42054 [ENOENT] O_CREAT is not set and the named shared memory object does not exist.

42055 [ENOSPC] There is insufficient space for the creation of the new shared memory object.

42056 EXAMPLES

42057 Creating and Mapping a Shared Memory Object

2

42058 The following code segment demonstrates the use of *shm_open()* to create a shared memory 2
42059 object which is then sized using *ftruncate()* before being mapped into the process address space 2
42060 using *mmap()*: 2

```
42061 #include <unistd.h> 2
42062 #include <sys/mman.h> 2
42063 ... 2
42064 #define MAX_LEN 10000 2
42065 struct region { /* Defines "structure" of shared memory */ 2
42066     int len; 2
42067     char buf[MAX_LEN]; 2
42068 }; 2
42069 struct region *rptr; 2
```

```

42070     int fd;                                2
42071     /* Create shared memory object and set its size */
42072     fd = shm_open( "/myregion", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR); 2
42073     if (fd == -1)                            2
42074         /* Handle error */;                  2
42075     if (ftruncate(fd, sizeof(struct region)) == -1) 2
42076         /* Handle error */;                  2
42077     /* Map shared memory object */           2
42078     rptr = mmap(NULL, sizeof(struct region),      2
42079         PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0); 2
42080     if (rptr == MAP_FAILED)                   2
42081         /* Handle error */;                  2
42082     /* Now we can refer to mapped region using fields of rptr;    2
42083         for example, rptr->len */          2
42084     ...

```

42085 APPLICATION USAGE

42086 None.

42087 RATIONALE

42088 When the Memory Mapped Files option is supported, the normal *open()* call is used to obtain a
 42089 descriptor to a file to be mapped according to existing practice with *mmap()*. When the Shared
 42090 Memory Objects option is supported, the *shm_open()* function shall obtain a descriptor to the
 42091 shared memory object to be mapped.

42092 There is ample precedent for having a file descriptor represent several types of objects. In the
 42093 POSIX.1-1990 standard, a file descriptor can represent a file, a pipe, a FIFO, a tty, or a directory.
 42094 Many implementations simply have an operations vector, which is indexed by the file descriptor
 42095 type and does very different operations. Note that in some cases the file descriptor passed to
 42096 generic operations on file descriptors is returned by *open()* or *creat()* and in some cases returned
 42097 by alternate functions, such as *pipe()*. The latter technique is used by *shm_open()*.

42098 Note that such shared memory objects can actually be implemented as mapped files. In both
 42099 cases, the size can be set after the open using *ftruncate()*. The *shm_open()* function itself does not
 42100 create a shared object of a specified size because this would duplicate an extant function that set
 42101 the size of an object referenced by a file descriptor.

42102 On implementations where memory objects are implemented using the existing file system, the
 42103 *shm_open()* function may be implemented using a macro that invokes *open()*, and the
 42104 *shm_unlink()* function may be implemented using a macro that invokes *unlink()*.

42105 For implementations without a permanent file system, the definition of the name of the memory
 42106 objects is allowed not to survive a system reboot. Note that this allows systems with a
 42107 permanent file system to implement memory objects as data structures internal to the
 42108 implementation as well.

42109 On implementations that choose to implement memory objects using memory directly, a
 42110 *shm_open()* followed by an *ftruncate()* and *close()* can be used to preallocate a shared memory
 42111 area and to set the size of that preallocation. This may be necessary for systems without virtual
 42112 memory hardware support in order to ensure that the memory is contiguous.

42113 The set of valid open flags to *shm_open()* was restricted to O_RDONLY, O_RDWR, O_CREAT,
 42114 and O_TRUNC because these could be easily implemented on most memory mapping systems.

42115 This volume of IEEE Std 1003.1-2001 is silent on the results if the implementation cannot supply
42116 the requested file access because of implementation-defined reasons, including hardware ones.

42117 The error conditions [EACCES] and [ENOTSUP] are provided to inform the application that the
42118 implementation cannot complete a request.

42119 [EACCES] indicates for implementation-defined reasons, probably hardware-related, that the
42120 implementation cannot comply with a requested mode because it conflicts with another
42121 requested mode. An example might be that an application desires to open a memory object two
42122 times, mapping different areas with different access modes. If the implementation cannot map a
42123 single area into a process space in two places, which would be required if different access modes
42124 were required for the two areas, then the implementation may inform the application at the time
42125 of the second open.

42126 [ENOTSUP] indicates for implementation-defined reasons, probably hardware-related, that the
42127 implementation cannot comply with a requested mode at all. An example would be that the
42128 hardware of the implementation cannot support write-only shared memory areas.

42129 On all implementations, it may be desirable to restrict the location of the memory objects to
42130 specific file systems for performance (such as a RAM disk) or implementation-defined reasons
42131 (shared memory supported directly only on certain file systems). The *shm_open()* function may
42132 be used to enforce these restrictions. There are a number of methods available to the application
42133 to determine an appropriate name of the file or the location of an appropriate directory. One
42134 way is from the environment via *getenv()*. Another would be from a configuration file.

42135 This volume of IEEE Std 1003.1-2001 specifies that memory objects have initial contents of zero
42136 when created. This is consistent with current behavior for both files and newly allocated
42137 memory. For those implementations that use physical memory, it would be possible that such
42138 implementations could simply use available memory and give it to the process uninitialized.
42139 This, however, is not consistent with standard behavior for the uninitialized data area, the stack,
42140 and of course, files. Finally, it is highly desirable to set the allocated memory to zero for security
42141 reasons. Thus, initializing memory objects to zero is required.

42142 FUTURE DIRECTIONS

42143 None.

42144 SEE ALSO

42145 *close()*, *dup()*, *exec*, *fcntl()*, *mmap()*, *shmat()*, *shmctl()*, *shmdt()*, *shm_unlink()*, *umask()*, the Base
42146 Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>, <sys/mman.h>

42147 CHANGE HISTORY

42148 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

42149 Issue 6

42150 The *shm_open()* function is marked as part of the Shared Memory Objects option.

42151 The [ENOSYS] error condition has been removed as stubs need not be provided if an
42152 implementation does not support the Shared Memory Objects option.

42153 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/126 is applied, adding the example to the EXAMPLES section. 2
42154 2

42155 NAME

42156 shm_unlink — remove a shared memory object (**REALTIME**)

42157 SYNOPSIS

42158 SHM `#include <sys/mman.h>`

42159 `int shm_unlink(const char *name);`

42160

42161 DESCRIPTION

42162 The *shm_unlink()* function shall remove the name of the shared memory object named by the
42163 string pointed to by *name*.

42164 If one or more references to the shared memory object exist when the object is unlinked, the
42165 name shall be removed before *shm_unlink()* returns, but the removal of the memory object
42166 contents shall be postponed until all open and map references to the shared memory object have
42167 been removed.

42168 Even if the object continues to exist after the last *shm_unlink()*, reuse of the name shall
42169 subsequently cause *shm_open()* to behave as if no shared memory object of this name exists (that
42170 is, *shm_open()* will fail if O_CREAT is not set, or will create a new shared memory object if
42171 O_CREAT is set).

42172 RETURN VALUE

42173 Upon successful completion, a value of zero shall be returned. Otherwise, a value of -1 shall be
42174 returned and *errno* set to indicate the error. If -1 is returned, the named shared memory object
42175 shall not be changed by this function call.

42176 ERRORS

42177 The *shm_unlink()* function shall fail if:

42178 [EACCES] Permission is denied to unlink the named shared memory object.

42179 [ENAMETOOLONG]

42180 The length of the *name* argument exceeds {PATH_MAX} or a pathname
42181 component is longer than {NAME_MAX}.

42182 [ENOENT] The named shared memory object does not exist.

42183 EXAMPLES

42184 None.

42185 APPLICATION USAGE

42186 Names of memory objects that were allocated with *open()* are deleted with *unlink()* in the usual
42187 fashion. Names of memory objects that were allocated with *shm_open()* are deleted with
42188 *shm_unlink()*. Note that the actual memory object is not destroyed until the last close and
42189 unmap on it have occurred if it was already in use.

42190 RATIONALE

42191 None.

42192 FUTURE DIRECTIONS

42193 None.

42194 SEE ALSO

42195 *close()*, *mmap()*, *munmap()*, *shmat()*, *shmctl()*, *shmdt()*, *shm_open()*, the Base Definitions volume
42196 of IEEE Std 1003.1-2001, <sys/mman.h>

42197 CHANGE HISTORY

42198 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

42199 Issue 6

42200 The *shm_unlink()* function is marked as part of the Shared Memory Objects option.

42201 In the DESCRIPTION, text is added to clarify that reusing the same name after a *shm_unlink()*
42202 will not attach to the old shared memory object.

42203 The [ENOSYS] error condition has been removed as stubs need not be provided if an
42204 implementation does not support the Shared Memory Objects option.

42205 NAME

42206 shmat — XSI shared memory attach operation

42207 SYNOPSIS

42208 XSI

```
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

42210

42211 DESCRIPTION

42212 The *shmat()* function operates on XSI shared memory (see the Base Definitions volume of
42213 IEEE Std 1003.1-2001, Section 3.340, Shared Memory Object). It is unspecified whether this
42214 function interoperates with the realtime interprocess communication facilities defined in Section
42215 2.8 (on page 41).

42216 The *shmat()* function attaches the shared memory segment associated with the shared memory
42217 identifier specified by *shmid* to the address space of the calling process. The segment is attached
42218 at the address specified by one of the following criteria:

- 42219 • If *shmaddr* is a null pointer, the segment is attached at the first available address as selected
42220 by the system.
- 42221 • If *shmaddr* is not a null pointer and (*shmflg* &SHM_RND) is non-zero, the segment is attached
42222 at the address given by (*shmaddr* -((*uintptr_t*)*shmaddr* %SHMLBA)). The character ' % ' is the
42223 C-language remainder operator.
- 42224 • If *shmaddr* is not a null pointer and (*shmflg* &SHM_RND) is 0, the segment is attached at the
42225 address given by *shmaddr*.
- 42226 • The segment is attached for reading if (*shmflg* &SHM_RDONLY) is non-zero and the calling
42227 process has read permission; otherwise, if it is 0 and the calling process has read and write
42228 permission, the segment is attached for reading and writing.

42229 RETURN VALUE

42230 Upon successful completion, *shmat()* shall increment the value of *shm_nattach* in the data
42231 structure associated with the shared memory ID of the attached shared memory segment and
42232 return the segment's start address.

42233 Otherwise, the shared memory segment shall not be attached, *shmat()* shall return -1, and *errno*
42234 shall be set to indicate the error.

42235 ERRORS

42236 The *shmat()* function shall fail if:

42237 [EACCES]	42238	Operation permission is denied to the calling process; see Section 2.7 (on page 39).
42239 [EINVAL]	42240	The value of <i>shmid</i> is not a valid shared memory identifier, the <i>shmaddr</i> is not a 42241 null pointer, and the value of (<i>shmaddr</i> -((<i>uintptr_t</i>) <i>shmaddr</i> %SHMLBA)) is an 42242 illegal address for attaching shared memory; or the <i>shmaddr</i> is not a null 42243 pointer, (<i>shmflg</i> &SHM_RND) is 0, and the value of <i>shmaddr</i> is an illegal address for attaching shared memory.
42244 [EMFILE]	42245	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.
42246 [ENOMEM]	42247	The available data space is not large enough to accommodate the shared memory segment.

42248 EXAMPLES

42249 None.

42250 APPLICATION USAGE

42251 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
42252 Application developers who need to use IPC should design their applications so that modules
42253 using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the
42254 alternative interfaces.

42255 RATIONALE

42256 None.

42257 FUTURE DIRECTIONS

42258 None.

42259 SEE ALSO

42260 Section 2.7 (on page 39), Section 2.8 (on page 41), *exec*, *exit()*, *fork()*, *shmctl()*, *shmdt()*, *shmget()*,
42261 *shm_open()*, *shm_unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/shm.h>

42262 CHANGE HISTORY

42263 First released in Issue 2. Derived from Issue 2 of the SVID.

42264 Issue 5

42265 Moved from SHARED MEMORY to BASE.

42266 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
42267 DIRECTIONS to a new APPLICATION USAGE section.

42268 Issue 6

42269 The Open Group Corrigendum U021/13 is applied.

42270 NAME

42271 shmctl — XSI shared memory control operations

42272 SYNOPSIS

42273 XSI #include <sys/shm.h>

42274 int shmctl(int *shmid*, int *cmd*, struct *shmid_ds* **buf*);

42275

42276 DESCRIPTION

42277 The *shmctl()* function operates on XSI shared memory (see the Base Definitions volume of
 42278 IEEE Std 1003.1-2001, Section 3.340, Shared Memory Object). It is unspecified whether this
 42279 function interoperates with the realtime interprocess communication facilities defined in Section
 42280 2.8 (on page 41).

42281 The *shmctl()* function provides a variety of shared memory control operations as specified by
 42282 *cmd*. The following values for *cmd* are available:

42283 IPC_STAT Place the current value of each member of the **shmid_ds** data structure
 42284 associated with *shmid* into the structure pointed to by *buf*. The contents of the
 42285 structure are defined in <sys/shm.h>.

42286 IPC_SET Set the value of the following members of the **shmid_ds** data structure
 42287 associated with *shmid* to the corresponding value found in the structure
 42288 pointed to by *buf*:

42289 shm_perm.uid
 42290 shm_perm.gid
 42291 shm_perm.mode Low-order nine bits.

42292 IPC_SET can only be executed by a process that has an effective user ID equal
 42293 to either that of a process with appropriate privileges or to the value of
 42294 *shm_perm.cuid* or *shm_perm.uid* in the **shmid_ds** data structure associated with
 42295 *shmid*.

42296 IPC_RMID Remove the shared memory identifier specified by *shmid* from the system and
 42297 destroy the shared memory segment and **shmid_ds** data structure associated
 42298 with it. IPC_RMID can only be executed by a process that has an effective user
 42299 ID equal to either that of a process with appropriate privileges or to the value
 42300 of *shm_perm.cuid* or *shm_perm.uid* in the **shmid_ds** data structure associated
 42301 with *shmid*.

42302 RETURN VALUE

42303 Upon successful completion, *shmctl()* shall return 0; otherwise, it shall return -1 and set *errno* to
 42304 indicate the error.

42305 ERRORS

42306 The *shmctl()* function shall fail if:

42307 [EACCES] The argument *cmd* is equal to IPC_STAT and the calling process does not have
 42308 read permission; see Section 2.7 (on page 39).

42309 [EINVAL] The value of *shmid* is not a valid shared memory identifier, or the value of *cmd*
 42310 is not a valid command.

42311 [EPERM] The argument *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID
 42312 of the calling process is not equal to that of a process with appropriate
 42313 privileges and it is not equal to the value of *shm_perm.cuid* or *shm_perm.uid* in
 42314 the data structure associated with *shmid*.

42315 The *shmctl()* function may fail if:

42316 [EOVERFLOW] The *cmd* argument is IPC_STAT and the *gid* or *uid* value is too large to be stored in the structure pointed to by the *buf* argument.
42317

42318 EXAMPLES

42319 None.

42320 APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the alternative interfaces.

42325 RATIONALE

42326 None.

42327 FUTURE DIRECTIONS

42328 None.

42329 SEE ALSO

42330 Section 2.7 (on page 39), Section 2.8 (on page 41), *shmat()*, *shmdt()*, *shmget()*, *shm_open()*,
42331 *shm_unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/shm.h>

42332 CHANGE HISTORY

42333 First released in Issue 2. Derived from Issue 2 of the SVID.

42334 Issue 5

42335 Moved from SHARED MEMORY to BASE.

42336 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
42337 DIRECTIONS to a new APPLICATION USAGE section.

42338 NAME

42339 shmdt — XSI shared memory detach operation

42340 SYNOPSIS

42341 XSI #include <sys/shm.h>

42342 int shmdt(const void *shmaddr);

42343

42344 DESCRIPTION

42345 The *shmdt()* function operates on XSI shared memory (see the Base Definitions volume of
42346 IEEE Std 1003.1-2001, Section 3.340, Shared Memory Object). It is unspecified whether this
42347 function interoperates with the realtime interprocess communication facilities defined in Section
42348 2.8 (on page 41).

42349 The *shmdt()* function detaches the shared memory segment located at the address specified by
42350 *shmaddr* from the address space of the calling process.

42351 RETURN VALUE

42352 Upon successful completion, *shmdt()* shall decrement the value of *shm_nattch* in the data
42353 structure associated with the shared memory ID of the attached shared memory segment and
42354 return 0.

42355 Otherwise, the shared memory segment shall not be detached, *shmdt()* shall return -1, and *errno*
42356 shall be set to indicate the error.

42357 ERRORS

42358 The *shmdt()* function shall fail if:

42359 [EINVAL] The value of *shmaddr* is not the data segment start address of a shared
42360 memory segment.

42361 EXAMPLES

42362 None.

42363 APPLICATION USAGE

42364 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
42365 Application developers who need to use IPC should design their applications so that modules
42366 using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the
42367 alternative interfaces.

42368 RATIONALE

42369 None.

42370 FUTURE DIRECTIONS

42371 None.

42372 SEE ALSO

42373 Section 2.7 (on page 39), Section 2.8 (on page 41), *exec*, *exit()*, *fork()*, *shmat()*, *shmctl()*, *shmget()*,
42374 *shm_open()*, *shm_unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/shm.h>

42375 CHANGE HISTORY

42376 First released in Issue 2. Derived from Issue 2 of the SVID.

42377 Issue 5

42378 Moved from SHARED MEMORY to BASE.

42379 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
42380 DIRECTIONS to a new APPLICATION USAGE section.

42381 NAME

42382 *shmget* — get an XSI shared memory segment

42383 SYNOPSIS

42384 XSI #include <sys/shm.h>

```
42385        int shmget(key_t key, size_t size, int shmfld);
```

42386

42387 DESCRIPTION

42388 The *shmget()* function operates on XSI shared memory (see the Base Definitions volume of
 42389 IEEE Std 1003.1-2001, Section 3.340, Shared Memory Object). It is unspecified whether this
 42390 function interoperates with the realtime interprocess communication facilities defined in Section
 42391 2.8 (on page 41).

42392 The *shmget()* function shall return the shared memory identifier associated with *key*.

42393 A shared memory identifier, associated data structure, and shared memory segment of at least
 42394 *size* bytes (see <sys/shm.h>) are created for *key* if one of the following is true:

- 42395 • The argument *key* is equal to IPC_PRIVATE.
- 42396 • The argument *key* does not already have a shared memory identifier associated with it and
 42397 (*shmfld* &IPC_CREAT) is non-zero.

42398 Upon creation, the data structure associated with the new shared memory identifier shall be
 42399 initialized as follows:

- 42400 • The values of *shm_perm.cuid*, *shm_perm.uid*, *shm_perm.cgid*, and *shm_perm.gid* are set equal to
 42401 the effective user ID and effective group ID, respectively, of the calling process.
- 42402 • The low-order nine bits of *shm_perm.mode* are set equal to the low-order nine bits of *shmfld*.
- 42403 • The value of *shm_segsz* is set equal to the value of *size*.
- 42404 • The values of *shm_lpid*, *shm_nattch*, *shm_atime*, and *shm_dtime* are set equal to 0.
- 42405 • The value of *shm_ctime* is set equal to the current time.

42406 When the shared memory segment is created, it shall be initialized with all zero values.

42407 RETURN VALUE

42408 Upon successful completion, *shmget()* shall return a non-negative integer, namely a shared
 42409 memory identifier; otherwise, it shall return -1 and set *errno* to indicate the error.

42410 ERRORS

42411 The *shmget()* function shall fail if:

- | | |
|-----------------------|--|
| 42412 [EACCES] | A shared memory identifier exists for <i>key</i> but operation permission as
42413 specified by the low-order nine bits of <i>shmfld</i> would not be granted; see
42414 Section 2.7 (on page 39). |
| 42415 [EEXIST] | A shared memory identifier exists for the argument <i>key</i> but (<i>shmfld</i>
42416 &IPC_CREAT) &&(<i>shmfld</i> &IPC_EXCL) is non-zero. |
| 42417 [EINVAL] | A shared memory segment is to be created and the value of <i>size</i> is less than
42418 the system-imposed minimum or greater than the system-imposed maximum. |
| 42419 [EINVAL] | No shared memory segment is to be created and a shared memory segment
42420 exists for <i>key</i> but the size of the segment associated with it is less than <i>size</i> and
42421 <i>size</i> is not 0. |

42422	[ENOENT]	A shared memory identifier does not exist for the argument <i>key</i> and (<i>shmflg</i> &IPC_CREAT) is 0.
42424	[ENOMEM]	A shared memory identifier and associated shared memory segment shall be created, but the amount of available physical memory is not sufficient to fill the request.
42427	[ENOSPC]	A shared memory identifier is to be created, but the system-imposed limit on the maximum number of allowed shared memory identifiers system-wide would be exceeded.

42430 EXAMPLES

42431 None.

42432 APPLICATION USAGE

42433 The POSIX Realtime Extension defines alternative interfaces for interprocess communication.
42434 Application developers who need to use IPC should design their applications so that modules
42435 using the IPC routines described in Section 2.7 (on page 39) can be easily modified to use the
42436 alternative interfaces.

42437 RATIONALE

42438 None.

42439 FUTURE DIRECTIONS

42440 None.

42441 SEE ALSO

42442 Section 2.7 (on page 39), Section 2.8 (on page 41), *shmat()*, *shmctl()*, *shmdt()*, *shm_open()*,
42443 *shm_unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/shm.h>

42444 CHANGE HISTORY

42445 First released in Issue 2. Derived from Issue 2 of the SVID.

42446 Issue 5

42447 Moved from SHARED MEMORY to BASE.

42448 The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
42449 DIRECTIONS to a new APPLICATION USAGE section.

42450 NAME

42451 **shutdown** — shut down socket send and receive operations

42452 SYNOPSIS

```
42453       #include <sys/socket.h>
42454       int shutdown(int socket, int how);
```

42455 DESCRIPTION

42456 The *shutdown()* function shall cause all or part of a full-duplex connection on the socket
42457 associated with the file descriptor *socket* to be shut down.

42458 The *shutdown()* function takes the following arguments:

42459 *socket* Specifies the file descriptor of the socket.

42460 *how* Specifies the type of shutdown. The values are as follows:

42461 SHUT_RD Disables further receive operations.

42462 SHUT_WR Disables further send operations.

42463 SHUT_RDWR Disables further send and receive operations.

42464 The *shutdown()* function disables subsequent send and/or receive operations on a socket,
42465 depending on the value of the *how* argument.

42466 RETURN VALUE

42467 Upon successful completion, *shutdown()* shall return 0; otherwise, -1 shall be returned and *errno*
42468 set to indicate the error.

42469 ERRORS

42470 The *shutdown()* function shall fail if:

42471 [EBADF] The *socket* argument is not a valid file descriptor.

42472 [EINVAL] The *how* argument is invalid.

42473 [ENOTCONN] The socket is not connected.

42474 [ENOTSOCK] The *socket* argument does not refer to a socket.

42475 The *shutdown()* function may fail if:

42476 [ENOBUFS] Insufficient resources were available in the system to perform the operation.

42477 EXAMPLES

42478 None.

42479 APPLICATION USAGE

42480 None.

42481 RATIONALE

42482 None.

42483 FUTURE DIRECTIONS

42484 None.

42485 SEE ALSO

42486 *getsockopt()*, *read()*, *recv()*, *recvfrom()*, *recvmsg()*, *select()*, *send()*, *sendto()*, *setsockopt()*, *socket()*,
42487 *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

42488 CHANGE HISTORY

42489 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

42490 NAME

42491 sigaction — examine and change a signal action

42492 SYNOPSIS

42493 CX #include <signal.h>

```
42494        int sigaction(int sig, const struct sigaction *restrict act,
42495                struct sigaction *restrict oact);
```

42496

42497 DESCRIPTION

42498 The **sigaction()** function allows the calling process to examine and/or specify the action to be
 42499 associated with a specific signal. The argument *sig* specifies the signal; acceptable values are
 42500 defined in <**signal.h**>.

42501 The structure **sigaction**, used to describe an action to be taken, is defined in the <**signal.h**>
 42502 header to include at least the following members:

42503

Member Type	Member Name	Description	
void(*)(int)	<i>sa_handler</i>	Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL.	1
sigset_t	<i>sa_mask</i>	Additional set of signals to be blocked during execution of signal-catching function.	1
int	<i>sa_flags</i>	Special flags to affect behavior of signal.	1
void(*)(int, siginfo_t *, void *)	<i>sa_sigaction</i>	Pointer to a signal-catching function.	1

42514 The storage occupied by *sa_handler* and *sa_sigaction* may overlap, and a conforming application
 42515 shall not use both simultaneously.

42516 If the argument *act* is not a null pointer, it points to a structure specifying the action to be
 42517 associated with the specified signal. If the argument *oact* is not a null pointer, the action
 42518 previously associated with the signal is stored in the location pointed to by the argument *oact*. If
 42519 the argument *act* is a null pointer, signal handling is unchanged; thus, the call can be used to
 42520 enquire about the current handling of a given signal. The SIGKILL and SIGSTOP signals shall
 42521 not be added to the signal mask using this mechanism; this restriction shall be enforced by the
 42522 system without causing an error to be indicated.

42523 If the SA_SIGINFO flag (see below) is cleared in the *sa_flags* field of the **sigaction** structure, the
 42524 XSI|RTS *sa_handler* field identifies the action to be associated with the specified signal. If the
 42525 SA_SIGINFO flag is set in the *sa_flags* field, and the implementation supports the Realtime
 42526 Signals Extension option or the XSI Extension option, the *sa_sigaction* field specifies a signal-
 42527 catching function.

2

42528 The *sa_flags* field can be used to modify the behavior of the specified signal.

42529 The following flags, defined in the <**signal.h**> header, can be set in *sa_flags*:

42530 XSI SA_NOCLDSTOP Do not generate SIGCHLD when children stop or stopped children
 42531 continue.

42532 If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is not set in *sa_flags*, and
 42533 the implementation supports the SIGCHLD signal, then a SIGCHLD
 42534 signal shall be generated for the calling process whenever any of its child
 42535 XSI

42536		processes stop and a SIGCHLD signal may be generated for the calling process whenever any of its stopped child processes are continued. If <i>sig</i> is SIGCHLD and the SA_NOCLDSTOP flag is set in <i>sa_flags</i> , then the implementation shall not generate a SIGCHLD signal in this way.
42537		
42538		
42539		
42540 XSI	SA_ONSTACK	If set and an alternate signal stack has been declared with <i>sigaltstack()</i> , the signal shall be delivered to the calling process on that stack. Otherwise, the signal shall be delivered on the current stack.
42541		
42542		
42543 XSI	SA_RESETHAND	If set, the disposition of the signal shall be reset to SIG_DFL and the SA_SIGINFO flag shall be cleared on entry to the signal handler.
42544		
42545		Note: SIGILL and SIGTRAP cannot be automatically reset when delivered; the system silently enforces this restriction.
42546		
42547		Otherwise, the disposition of the signal shall not be modified on entry to the signal handler.
42548		
42549		In addition, if this flag is set, <i>sigaction()</i> behaves as if the SA_NODEFER flag were also set.
42550		
42551 XSI	SA_RESTART	This flag affects the behavior of interruptible functions; that is, those specified to fail with <i>errno</i> set to [EINTR]. If set, and a function specified as interruptible is interrupted by this signal, the function shall restart and shall not fail with [EINTR] unless otherwise specified. If the flag is not set, interruptible functions interrupted by this signal shall fail with <i>errno</i> set to [EINTR].
42552		
42553		
42554		
42555		
42556		
42557	SA_SIGINFO	If cleared and the signal is caught, the signal-catching function shall be entered as:
42558		
42559		<pre>void func(int signo);</pre>
42560		where <i>signo</i> is the only argument to the signal-catching function. In this case, the application shall use the <i>sa_handler</i> member to describe the signal-catching function and the application shall not modify the <i>sa_sigaction</i> member.
42561		
42562		
42563		
42564 XSI RTS		If SA_SIGINFO is set and the signal is caught, the signal-catching function shall be entered as:
42565		
42566		<pre>void func(int signo, siginfo_t *info, void *context);</pre>
42567		where two additional arguments are passed to the signal-catching function. The second argument shall point to an object of type siginfo_t explaining the reason why the signal was generated; the third argument can be cast to a pointer to an object of type ucontext_t to refer to the receiving thread's context that was interrupted when the signal was delivered. In this case, the application shall use the <i>sa_sigaction</i> member to describe the signal-catching function and the application shall not modify the <i>sa_handler</i> member.
42568		
42569		
42570		
42571		
42572		
42573		
42574		
42575		The <i>si_signo</i> member contains the system-generated signal number.
42576 XSI		The <i>si_errno</i> member may contain implementation-defined additional error information; if non-zero, it contains an error number identifying the condition that caused the signal to be generated.
42577		
42578		
42579 XSI RTS		The <i>si_code</i> member contains a code identifying the cause of the signal.
		2 2

42580 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 <signal.h> header description contains information about the signal-specific contents of the elements of the siginfo_t type.
42585 XSI	SA_NOCLDWAIT	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.
42593 XSI	SA_NODEFER	If set and <i>sig</i> is caught, <i>sig</i> shall not be added to the thread's 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 thread's signal mask on entry to the signal handler.
42597		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.
42603		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.
42606		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.
42611		If <i>sigaction()</i> fails, no new signal handler is installed.
42612		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].
42614	RTS	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.
42623		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.

42625 RETURN VALUE

42626 Upon successful completion, *sigaction()* shall return 0; otherwise, -1 shall be returned, *errno* shall
42627 be set to indicate the error, and no new signal-catching function shall be installed.

42628 ERRORS

42629 The *sigaction()* function shall fail if:

- | | |
|-----------------|---|
| 42630 [EINVAL] | The <i>sig</i> argument is not a valid signal number or an attempt is made to catch a
42631 signal that cannot be caught or ignore a signal that cannot be ignored. |
| 42632 [ENOTSUP] | The SA_SIGINFO bit flag is set in the <i>sa_flags</i> field of the sigaction structure,
42633 and the implementation does not support either the Realtime Signals
42634 Extension option, or the XSI Extension option. |

42635 The *sigaction()* function may fail if:

- | | |
|----------------|--|
| 42636 [EINVAL] | An attempt was made to set the action to SIG_DFL for a signal that cannot be
42637 caught or ignored (or both). |
|----------------|--|

42638 EXAMPLES**42639 Establishing a Signal Handler**

2

42640 The following example demonstrates the use of *sigaction()* to establish a handler for the SIGINT
42641 signal.

```
42642 #include <signal.h>
42643 static void handler(int signum)
42644 {
42645     /* Take appropriate actions for signal delivery */
42646 }
42647 int main()
42648 {
42649     struct sigaction sa;
42650
42651     sa.sa_handler = handler;
42652     sigemptyset(&sa.sa_mask);
42653     sa.sa_flags = SA_RESTART; /* Restart functions if
42654                                interrupted by handler */
42655     if (sigaction(SIGINT, &sa, NULL) == -1)
42656         /* Handle error */;
42657
42658     /* Further code */
42659 }
```

42658 APPLICATION USAGE

42659 The *sigaction()* function supersedes the *signal()* function, and should be used in preference. In
42660 particular, *sigaction()* and *signal()* should not be used in the same process to control the same
42661 signal. The behavior of reentrant functions, as defined in the DESCRIPTION, is as specified by
42662 this volume of IEEE Std 1003.1-2001, regardless of invocation from a signal-catching function.
42663 This is the only intended meaning of the statement that reentrant functions may be used in
42664 signal-catching functions without restrictions. Applications must still consider all effects of such
42665 functions on such things as data structures, files, and process state. In particular, application
42666 writers need to consider the restrictions on interactions when interrupting *sleep()* and
42667 interactions among multiple handles for a file description. The fact that any specific function is
42668 listed as reentrant does not necessarily mean that invocation of that function from a signal-
42669 catching function is recommended.

In order to prevent errors arising from interrupting non-reentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore (see *semget()*, *sem_init()*, *sem_open()*, and so on). Note in particular that even the “safe” functions may modify *errno*; the signal-catching function, if not executing as an independent thread, may want to save and restore its value. Naturally, the same principles apply to the reentrancy of application routines and asynchronous data access. Note that *longjmp()* and *siglongjmp()* are not in the list of reentrant functions. This is because the code executing after *longjmp()* and *siglongjmp()* can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use *longjmp()* and *siglongjmp()* from within signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either *malloc()* or *free()* functions or the standard I/O library, both of which traditionally use data structures in a non-reentrant manner. Since any combination of different functions using a common data structure can cause reentrancy problems, this volume of IEEE Std 1003.1-2001 does not define the behavior when any unsafe function is called in a signal handler that interrupts an unsafe function.

If the signal occurs other than as the result of calling *abort()*, *kill()*, or *raise()*, the behavior is undefined if the signal handler calls any function in the standard library other than one of the functions listed in the table above or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type **volatile sig_atomic_t**. 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 thread 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. 2 2

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:

```
42702 void handler(int sig, int code, struct sigcontext *scp,
42703     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.

42708 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

42718 functions that repeatedly call the *waitpid()* function with the WNOHANG flag set, acting on
42719 each child for which status is returned, until *waitpid()* returns zero. If stopped children are not of
42720 interest, the use of the SA_NOCLDSTOP flag can prevent the overhead from invoking the
42721 signal-catching routine when they stop.

42722 Some historical implementations also define other mechanisms for stopping processes, such as
42723 the *ptrace()* function. These implementations usually do not generate a SIGCHLD signal when
42724 processes stop due to this mechanism; however, that is beyond the scope of this volume of
42725 IEEE Std 1003.1-2001.

42726 This volume of IEEE Std 1003.1-2001 requires that calls to *sigaction()* that supply a NULL *act*
42727 argument succeed, even in the case of signals that cannot be caught or ignored (that is, SIGKILL
42728 or SIGSTOP). The System V *signal()* and BSD *sigvec()* functions return [EINVAL] in these cases
42729 and, in this respect, their behavior varies from *sigaction()*.

42730 This volume of IEEE Std 1003.1-2001 requires that *sigaction()* properly save and restore a signal
42731 action set up by the ISO C standard *signal()* function. However, there is no guarantee that the
42732 reverse is true, nor could there be given the greater amount of information conveyed by the
42733 **sigaction** structure. Because of this, applications should avoid using both functions for the same
42734 signal in the same process. Since this cannot always be avoided in case of general-purpose
42735 library routines, they should always be implemented with *sigaction()*.

42736 It was intended that the *signal()* function should be implementable as a library routine using
42737 *sigaction()*.

42738 The POSIX Realtime Extension extends the *sigaction()* function as specified by the POSIX.1-1990
42739 standard to allow the application to request on a per-signal basis via an additional signal action
42740 flag that the extra parameters, including the application-defined signal value, if any, be passed
42741 to the signal-catching function.

42742 FUTURE DIRECTIONS

42743 None.

42744 SEE ALSO

42745 Section 2.4 (on page 28), *bsd_signal()*, *kill()*, *_longjmp()*, *longjmp()*, *raise()*, *semget()*, *sem_init()*,
42746 *sem_open()*, *sigaddset()*, *sigaltstack()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *signal()*,
42747 *sigprocmask()*, *sigsuspend()*, *wait()*, *waitid()*, *waitpid()*, the Base Definitions volume of
42748 IEEE Std 1003.1-2001, <**signal.h**>, <**ucontext.h**>

42749 CHANGE HISTORY

42750 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

42751 Issue 5

42752 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and POSIX
42753 Threads Extension.

42754 In the DESCRIPTION, the second argument to *func* when SA_SIGINFO is set is no longer
42755 permitted to be NULL, and the description of permitted **siginfo_t** contents is expanded by
42756 reference to <**signal.h**>.

42757 Since the X/OPEN UNIX Extension functionality is now folded into the BASE, the [ENOTSUP]
42758 error is deleted.

42759 Issue 6

42760 The Open Group Corrigendum U028/7 is applied. In the paragraph entitled “Signal Effects on
42761 Other Functions”, a reference to *sigpending()* is added.

42762 In the DESCRIPTION, the text “Signal Generation and Delivery”, “Signal Actions”, and “Signal
42763 Effects on Other Functions” are moved to a separate section of this volume of

42764	IEEE Std 1003.1-2001.	
42765	Text describing functionality from the Realtime Signals option is marked.	
42766	The following changes are made for alignment with the ISO POSIX-1: 1996 standard:	
42767	• The [ENOTSUP] error condition is added.	
42768	The DESCRIPTION is updated to avoid use of the term “must” for application requirements.	
42769	The restrict keyword is added to the <i>sigaction()</i> prototype for alignment with the ISO/IEC 9899: 1999 standard.	
42771	References to the <i>wait3()</i> function are removed.	
42772	The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an extension over the ISO C standard.	
42774	IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/57 is applied, changing text in the table	1
42775	describing the sigaction structure.	1
42776	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/127 is applied, removing text from the	2
42777	DESCRIPTION duplicated later in the same section.	2
42778	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/128 is applied, updating the	2
42779	DESCRIPTION and APPLICATION USAGE sections. Changes are made to refer to the thread	2
42780	rather than the process.	2
42781	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/129 is applied, adding the example to the	2
42782	EXAMPLES section.	2

42783 NAME

42784 sigaddset — add a signal to a signal set

42785 SYNOPSIS

42786 CX #include <signal.h>
42787 int sigaddset(sigset_t *set, int signo);
42788

42789 DESCRIPTION

42790 The *sigaddset()* function adds the individual signal specified by the *signo* to the signal set pointed to by *set*.

42792 Applications shall call either *sigemptyset()* or *sigfillset()* at least once for each object of type *sigset_t* prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of *pthread_sigmask()*, *sigaction()*, *sigaddset()*, *sigdelset()*, *sigismember()*, *sigpending()*, *sigprocmask()*, *sigsuspend()*, *sigtimedwait()*, *sigwait()*, or *sigwaitinfo()*, the results are undefined.

42797 RETURN VALUE

42798 Upon successful completion, *sigaddset()* shall return 0; otherwise, it shall return -1 and set *errno* to indicate the error.

42800 ERRORS

42801 The *sigaddset()* function may fail if:

42802 [EINVAL] The value of the *signo* argument is an invalid or unsupported signal number.

42803 EXAMPLES

42804 None.

42805 APPLICATION USAGE

42806 None.

42807 RATIONALE

42808 None.

42809 FUTURE DIRECTIONS

42810 None.

42811 SEE ALSO

42812 Section 2.4 (on page 28), *sigaction()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*,
42813 *sigpending()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001,
42814 <signal.h>

42815 CHANGE HISTORY

42816 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

42817 Issue 5

42818 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
42819 previous issues.

42820 Issue 6

42821 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

42822 The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an
42823 extension over the ISO C standard.

42824 NAME

42825 sigaltstack — set and get signal alternate stack context

42826 SYNOPSIS

42827 XSI #include <signal.h>
42828 int sigaltstack(const stack_t *restrict ss, stack_t *restrict oss);
42829

42830 DESCRIPTION

42831 The *sigaltstack()* function allows a process to define and examine the state of an alternate stack 1
42832 for signal handlers for the current thread. Signals that have been explicitly declared to execute 1
42833 on the alternate stack shall be delivered on the alternate stack.

42834 If *ss* is not a null pointer, it points to a **stack_t** structure that specifies the alternate signal stack
42835 that shall take effect upon return from *sigaltstack()*. The *ss_flags* member specifies the new stack
42836 state. If it is set to SS_DISABLE, the stack is disabled and *ss_sp* and *ss_size* are ignored.
42837 Otherwise, the stack shall be enabled, and the *ss_sp* and *ss_size* members specify the new address
42838 and size of the stack.

42839 The range of addresses starting at *ss_sp* up to but not including *ss_sp+ss_size* is available to the
42840 implementation for use as the stack. This function makes no assumptions regarding which end
42841 is the stack base and in which direction the stack grows as items are pushed.

42842 If *oss* is not a null pointer, on successful completion it shall point to a **stack_t** structure that
42843 specifies the alternate signal stack that was in effect prior to the call to *sigaltstack()*. The *ss_sp*
42844 and *ss_size* members specify the address and size of that stack. The *ss_flags* member specifies the
42845 stack's state, and may contain one of the following values:

42846 SS_ONSTACK The process is currently executing on the alternate signal stack. Attempts to
42847 modify the alternate signal stack while the process is executing on it fail. This
42848 flag shall not be modified by processes.

42849 SS_DISABLE The alternate signal stack is currently disabled.

42850 The value SIGSTKSZ is a system default specifying the number of bytes that would be used to
42851 cover the usual case when manually allocating an alternate stack area. The value MINSIGSTKSZ
42852 is defined to be the minimum stack size for a signal handler. In computing an alternate stack
42853 size, a program should add that amount to its stack requirements to allow for the system
42854 implementation overhead. The constants SS_ONSTACK, SS_DISABLE, SIGSTKSZ, and
42855 MINSIGSTKSZ are defined in <signal.h>.

42856 After a successful call to one of the *exec* functions, there are no alternate signal stacks in the new
42857 process image.

42858 In some implementations, a signal (whether or not indicated to execute on the alternate stack)
42859 shall always execute on the alternate stack if it is delivered while another signal is being caught
42860 using the alternate stack.

42861 Use of this function by library threads that are not bound to kernel-scheduled entities results in
42862 undefined behavior.

42863 RETURN VALUE

42864 Upon successful completion, *sigaltstack()* shall return 0; otherwise, it shall return -1 and set *errno*
42865 to indicate the error.

42866 ERRORS

- 42867 The *sigaltstack()* function shall fail if:
- 42868 [EINVAL] The *ss* argument is not a null pointer, and the *ss_flags* member pointed to by *ss* contains flags other than SS_DISABLE.
- 42870 [ENOMEM] The size of the alternate stack area is less than MINSIGSTKSZ.
- 42871 [EPERM] An attempt was made to modify an active stack.

42872 EXAMPLES

42873 Allocating Memory for an Alternate Stack

42874 The following example illustrates a method for allocating memory for an alternate stack.

```
42875 #include <signal.h>
42876 ...
42877 if ((sigstk.ss_sp = malloc(SIGSTKSZ)) == NULL)
42878     /* Error return. */
42879     sigstk.ss_size = SIGSTKSZ;
42880     sigstk.ss_flags = 0;
42881     if (sigaltstack(&sigstk, (stack_t *)0) < 0)
42882         perror("sigaltstack");
```

42883 APPLICATION USAGE

42884 On some implementations, stack space is automatically extended as needed. On those implementations, automatic extension is typically not available for an alternate stack. If the stack overflows, the behavior is undefined.

42887 RATIONALE

42888 None.

42889 FUTURE DIRECTIONS

42890 None.

42891 SEE ALSO

42892 Section 2.4 (on page 28), *sigaction()*, *sigsetjmp()*, the Base Definitions volume of
42893 IEEE Std 1003.1-2001, <**signal.h**>

42894 CHANGE HISTORY

42895 First released in Issue 4, Version 2.

42896 Issue 5

42897 Moved from X/OPEN UNIX extension to BASE.

42898 The last sentence of the DESCRIPTION was included as an APPLICATION USAGE note in
42899 previous issues.

42900 Issue 6

42901 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

42902 The **restrict** keyword is added to the *sigaltstack()* prototype for alignment with the
42903 ISO/IEC 9899:1999 standard.

42904 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/58 is applied, updating the first sentence 1
42905 to include “for the current thread”. 1

42906 NAME

42907 *sigdelset* — delete a signal from a signal set

42908 SYNOPSIS

42909 CX

```
#include <signal.h>
```

42910

```
int sigdelset(sigset_t *set, int signo);
```

42911

42912 DESCRIPTION

42913 The *sigdelset()* function deletes the individual signal specified by *signo* from the signal set pointed to by *set*.

42915 Applications should call either *sigemptyset()* or *sigfillset()* at least once for each object of type *sigset_t* prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of *pthread_sigmask()*, *sigaction()*, *sigaddset()*, *sigdelset()*, *sigismember()*, *sigpending()*, *sigprocmask()*, *sigsuspend()*, *sigtimedwait()*, *sigwait()*, or *sigwaitinfo()*, the results are undefined.

42920 RETURN VALUE

42921 Upon successful completion, *sigdelset()* shall return 0; otherwise, it shall return -1 and set *errno* to indicate the error.

42923 ERRORS

42924 The *sigdelset()* function may fail if:

42925 [EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal number.

42927 EXAMPLES

42928 None.

42929 APPLICATION USAGE

42930 None.

42931 RATIONALE

42932 None.

42933 FUTURE DIRECTIONS

42934 None.

42935 SEE ALSO

42936 Section 2.4 (on page 28), *sigaction()*, *sigaddset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*,
42937 *sigpending()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001,
42938 <*signal.h*>

42939 CHANGE HISTORY

42940 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

42941 Issue 5

42942 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
42943 previous issues.

42944 Issue 6

42945 The SYNOPSIS is marked CX since the presence of this function in the <*signal.h*> header is an
42946 extension over the ISO C standard.

42947 NAME

42948 *sigemptyset* — initialize and empty a signal set

42949 SYNOPSIS

42950 CX `#include <signal.h>`

42951 `int sigemptyset(sigset_t *set);`

42952

42953 DESCRIPTION

42954 The *sigemptyset()* function initializes the signal set pointed to by *set*, such that all signals defined
42955 in IEEE Std 1003.1-2001 are excluded.

42956 RETURN VALUE

42957 Upon successful completion, *sigemptyset()* shall return 0; otherwise, it shall return -1 and set
42958 *errno* to indicate the error.

42959 ERRORS

42960 No errors are defined.

42961 EXAMPLES

42962 None.

42963 APPLICATION USAGE

42964 None.

42965 RATIONALE

42966 The implementation of the *sigemptyset()* (or *sigfillset()*) function could quite trivially clear (or
42967 set) all the bits in the signal set. Alternatively, it would be reasonable to initialize part of the
42968 structure, such as a version field, to permit binary-compatibility between releases where the size
42969 of the set varies. For such reasons, either *sigemptyset()* or *sigfillset()* must be called prior to any
42970 other use of the signal set, even if such use is read-only (for example, as an argument to
42971 *sigpending()*). This function is not intended for dynamic allocation.

42972 The *sigfillset()* and *sigemptyset()* functions require that the resulting signal set include (or
42973 exclude) all the signals defined in this volume of IEEE Std 1003.1-2001. Although it is outside the
42974 scope of this volume of IEEE Std 1003.1-2001 to place this requirement on signals that are
42975 implemented as extensions, it is recommended that implementation-defined signals also be
42976 affected by these functions. However, there may be a good reason for a particular signal not to
42977 be affected. For example, blocking or ignoring an implementation-defined signal may have
42978 undesirable side effects, whereas the default action for that signal is harmless. In such a case, it
42979 would be preferable for such a signal to be excluded from the signal set returned by *sigfillset()*.

42980 In early proposals there was no distinction between invalid and unsupported signals (the names
42981 of optional signals that were not supported by an implementation were not defined by that
42982 implementation). The [EINVAL] error was thus specified as a required error for invalid signals.
42983 With that distinction, it is not necessary to require implementations of these functions to
42984 determine whether an optional signal is actually supported, as that could have a significant
42985 performance impact for little value. The error could have been required for invalid signals and
42986 optional for unsupported signals, but this seemed unnecessarily complex. Thus, the error is
42987 optional in both cases.

42988 FUTURE DIRECTIONS

42989 None.

42990 SEE ALSO

42991 Section 2.4 (on page 28), *sigaction()*, *sigaddset()*, *sigdelset()*, *sigfillset()*, *sigismember()*, *sigpending()*,
42992 *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>

42993 CHANGE HISTORY

42994 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

42995 Issue 6

42996 The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an
42997 extension over the ISO C standard.

42998 NAME

42999 *sigfillset* — initialize and fill a signal set

43000 SYNOPSIS

43001 CX

```
#include <signal.h>
```

43002

```
int sigfillset(sigset_t *set);
```

43003

43004 DESCRIPTION

43005 The *sigfillset()* function shall initialize the signal set pointed to by *set*, such that all signals
43006 defined in this volume of IEEE Std 1003.1-2001 are included.

43007 RETURN VALUE

43008 Upon successful completion, *sigfillset()* shall return 0; otherwise, it shall return -1 and set *errno*
43009 to indicate the error.

43010 ERRORS

43011 No errors are defined.

43012 EXAMPLES

43013 None.

43014 APPLICATION USAGE

43015 None.

43016 RATIONALE

43017 Refer to *sigemptyset()* (on page 1373).

43018 FUTURE DIRECTIONS

43019 None.

43020 SEE ALSO

43021 Section 2.4 (on page 28), *sigaction()*, *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigismember()*,
43022 *sigpending()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of IEEE Std 1003.1-2001,
43023 <**signal.h**>

43024 CHANGE HISTORY

43025 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

43026 Issue 6

43027 The SYNOPSIS is marked CX since the presence of this function in the <**signal.h**> header is an
43028 extension over the ISO C standard.

43029 NAME

43030 *sighold*, *sigignore*, *sigpause*, *sigrelse*, *sigset* — signal management

43031 SYNOPSIS

43032 XSI #include <signal.h>

```
43033     int sighold(int sig);
43034     int sigignore(int sig);
43035     int sigpause(int sig);
43036     int sigrelse(int sig);
43037     void (*sigset(int sig, void (*disp)(int)))(int);
```

43038

43039 DESCRIPTION

43040 Use of any of these functions is unspecified in a multi-threaded process.

43041 The *sighold*(*sig*), *sigignore*(*sig*), *sigpause*(*sig*), *sigrelse*(*sig*), and *sigset*(*sig*) functions provide simplified signal management.

43043 The *sigset*(*sig*) 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*(*sig*) 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*(*sig*) 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*(*sig*) is used, and *disp* is not equal to SIG_HOLD, *sig* shall be removed from the calling process' signal mask.

43052 The *sighold*(*sig*) function shall add *sig* to the calling process' signal mask.

43053 The *sigrelse*(*sig*) function shall remove *sig* from the calling process' signal mask.

43054 The *sigignore*(*sig*) function shall set the disposition of *sig* to SIG_IGN.

43055 The *sigpause*(*sig*) function shall remove *sig* from the calling process' signal mask and suspend the calling process until a signal is received. The *sigpause*(*sig*) function shall restore the process' signal mask to its original state before returning.

43058 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*(*pid*), *waitid*(*id*), and *waitpid*(*pid*) shall fail and set *errno* to [ECHILD].

43063 RETURN VALUE

43064 Upon successful completion, *sigset*(*sig*) 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.

43067 The *sigpause*(*sig*) function shall suspend execution of the thread until a signal is received, whereupon it shall return -1 and set *errno* to [EINTR].

43069 For all other functions, upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

43071 ERRORS

43072 These functions shall fail if:

43073 [EINVAL] The *sig* argument is an illegal signal number.

43074 The *sigset()* and *sigignore()* functions shall fail if:

43075 [EINVAL] An attempt is made to catch a signal that cannot be caught, or to ignore a
43076 signal that cannot be ignored.

43077 EXAMPLES

43078 None.

43079 APPLICATION USAGE

43080 The *sigaction()* function provides a more comprehensive and reliable mechanism for controlling
43081 signals; new applications should use *sigaction()* rather than *sigset()*.

43082 The *sighold()* function, in conjunction with *sigrelse()* or *sigpause()*, may be used to establish
43083 critical regions of code that require the delivery of a signal to be temporarily deferred.

43084 The *sigsuspend()* function should be used in preference to *sigpause()* for broader portability.

43085 RATIONALE

43086 None.

43087 FUTURE DIRECTIONS

43088 None.

43089 SEE ALSO

43090 Section 2.4 (on page 28), *exec*, *pause()*, *sigaction()*, *signal()*, *sigsuspend()*, *waitid()*, the Base
43091 Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>

43092 CHANGE HISTORY

43093 First released in Issue 4, Version 2.

43094 Issue 5

43095 Moved from X/OPEN UNIX extension to BASE.

43096 The DESCRIPTION is updated to indicate that the *sigpause()* function restores the process'
43097 signal mask to its original state before returning.

43098 The RETURN VALUE section is updated to indicate that the *sigpause()* function suspends
43099 execution of the process until a signal is received, whereupon it returns -1 and sets *errno* to
43100 [EINTR].

43101 Issue 6

43102 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

43103 References to the *wait3()* function are removed.

43104 The XSI functions are split out into their own reference page.

43105 NAME

43106 *siginterrupt* — allow signals to interrupt functions

43107 SYNOPSIS

43108 XSI `#include <signal.h>`

43109 `int siginterrupt(int sig, int flag);`

43110

43111 DESCRIPTION

43112 The *siginterrupt()* function shall change the restart behavior when a function is interrupted by
43113 the specified signal. The function *siginterrupt(sig, flag)* has an effect as if implemented as:

```
43114        int siginterrupt(int sig, int flag) {  
43115            int ret;  
43116            struct sigaction act;  
43117            (void) sigaction(sig, NULL, &act);  
43118            if (flag)  
43119                act.sa_flags &= ~SA_RESTART;  
43120            else  
43121                act.sa_flags |= SA_RESTART;  
43122            ret = sigaction(sig, &act, NULL);  
43123            return ret;  
43124        }
```

1
1

43125 RETURN VALUE

43126 Upon successful completion, *siginterrupt()* shall return 0; otherwise, -1 shall be returned and
43127 *errno* set to indicate the error.

43128 ERRORS

43129 The *siginterrupt()* function shall fail if:

43130 [EINVAL] The *sig* argument is not a valid signal number.

43131 EXAMPLES

43132 None.

43133 APPLICATION USAGE

43134 The *siginterrupt()* function supports programs written to historical system interfaces. A
43135 conforming application, when being written or rewritten, should use *sigaction()* with the
43136 SA_RESTART flag instead of *siginterrupt()*.

43137 RATIONALE

43138 None.

43139 FUTURE DIRECTIONS

43140 None.

43141 SEE ALSO

43142 Section 2.4 (on page 28), *sigaction()*, the Base Definitions volume of IEEE Std 1003.1-2001,
43143 *<signal.h>*

43144 CHANGE HISTORY

43145 First released in Issue 4, Version 2.

43146 Issue 5

43147 Moved from X/OPEN UNIX extension to BASE.

43148 Issue 6

43149 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/59 is applied, correcting the declaration in
43150 the sample implementation given in the DESCRIPTION. 1 1

43151 NAME

43152 *sigismember* — test for a signal in a signal set

43153 SYNOPSIS

43154 CX

```
#include <signal.h>
```

43155

```
int sigismember(const sigset_t *set, int signo);
```

43156

43157 DESCRIPTION

43158 The *sigismember*() function shall test whether the signal specified by *signo* is a member of the set pointed to by *set*.

43160 Applications should call either *sigemptyset*() or *sigfillset*() at least once for each object of type *sigset_t* prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of *pthread_sigmask*(), *sigaction*(), *sigaddset*(), *sigdelset*(), *sigismember*(), *sigpending*(), *sigprocmask*(), *sigsuspend*(), *sigtimedwait*(), *sigwait*(), or *sigwaitinfo*(), the results are undefined.

43165 RETURN VALUE

43166 Upon successful completion, *sigismember*() shall return 1 if the specified signal is a member of the specified set, or 0 if it is not. Otherwise, it shall return -1 and set *errno* to indicate the error.

43168 ERRORS

43169 The *sigismember*() function may fail if:

43170 [EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal number.

43172 EXAMPLES

43173 None.

43174 APPLICATION USAGE

43175 None.

43176 RATIONALE

43177 None.

43178 FUTURE DIRECTIONS

43179 None.

43180 SEE ALSO

43181 Section 2.4 (on page 28), *sigaction*(), *sigaddset*(), *sigdelset*(), *sigfillset*(), *sigemptyset*(), *sigpending*(),
43182 *sigprocmask*(), *sigsuspend*(), the Base Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>

43183 CHANGE HISTORY

43184 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

43185 Issue 5

43186 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
43187 previous issues.

43188 Issue 6

43189 The SYNOPSIS is marked CX since the presence of this function in the <**signal.h**> header is an
43190 extension over the ISO C standard.

43191 NAME

43192 `siglongjmp` — non-local goto with signal handling

43193 SYNOPSIS

43194 CX

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


43195

```
void siglongjmp(sigjmp_buf env, int val);
```


43196

43197 DESCRIPTION

43198 The `siglongjmp()` function shall be equivalent to the `longjmp()` function, except as follows:

- 43199 • References to `setjmp()` shall be equivalent to `sigsetjmp()`.
- 43200 • The `siglongjmp()` function shall restore the saved signal mask if and only if the `env` argument
43201 was initialized by a call to `sigsetjmp()` with a non-zero `savemask` argument.

43202 RETURN VALUE

43203 After `siglongjmp()` is completed, program execution shall continue as if the corresponding
43204 invocation of `sigsetjmp()` had just returned the value specified by `val`. The `siglongjmp()` function
43205 shall not cause `sigsetjmp()` to return 0; if `val` is 0, `sigsetjmp()` shall return the value 1.

43206 ERRORS

43207 No errors are defined.

43208 EXAMPLES

43209 None.

43210 APPLICATION USAGE

43211 The distinction between `setjmp()` or `longjmp()` and `sigsetjmp()` or `siglongjmp()` is only significant
43212 for programs which use `sigaction()`, `sigprocmask()`, or `sigsuspend()`.

43213 RATIONALE

43214 None.

43215 FUTURE DIRECTIONS

43216 None.

43217 SEE ALSO

43218 `longjmp()`, `setjmp()`, `sigprocmask()`, `sigsetjmp()`, `sigsuspend()`, the Base Definitions volume of
43219 IEEE Std 1003.1-2001, `<setjmp.h>`

43220 CHANGE HISTORY

43221 First released in Issue 3. Included for alignment with the ISO POSIX-1 standard.

43222 Issue 5

43223 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

43224 Issue 6

43225 The DESCRIPTION is rewritten in terms of `longjmp()`.

43226 The SYNOPSIS is marked CX since the presence of this function in the `<setjmp.h>` header is an
43227 extension over the ISO C standard.

43228 NAME

43229 signal — signal management

43230 SYNOPSIS

43231 #include <signal.h>
43232 void (*signal(int *sig*, void (**func*)(int)))(int);

43233 DESCRIPTION

43234 CX 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.

43237 CX Use of this function is unspecified in a multi-threaded process.

43238 The *signal()* function chooses one of three ways in which receipt of the signal number *sig* is to be
43239 subsequently handled. If the value of *func* is SIG_DFL, default handling for that signal shall
43240 occur. If the value of *func* is SIG_IGN, the signal shall be ignored. Otherwise, the application
43241 shall ensure that *func* points to a function to be called when that signal occurs. An invocation of
43242 such a function because of a signal, or (recursively) of any further functions called by that
43243 invocation (other than functions in the standard library), is called a “signal handler”.

43244 When a signal occurs, and *func* points to a function, it is implementation-defined whether the
43245 equivalent of a:

43246 `signal(sig, SIG_DFL);`

43247 is executed or the implementation prevents some implementation-defined set of signals (at least
43248 including *sig*) from occurring until the current signal handling has completed. (If the value of *sig*
43249 is SIGILL, the implementation may alternatively define that no action is taken.) Next the
43250 equivalent of:

43251 `(*func)(sig);`

43252 is executed. If and when the function returns, if the value of *sig* was SIGFPE, SIGILL, or
43253 SIGSEGV or any other implementation-defined value corresponding to a computational
43254 exception, the behavior is undefined. Otherwise, the program shall resume execution at the
43255 CX point it was interrupted. If the signal occurs as the result of calling the *abort()*, *raise()*, *kill()*,
43256 *pthread_kill()*, or *sigqueue()* function, the signal handler shall not call the *raise()* function.

43257 CX If the signal occurs other than as the result of calling *abort()*, *raise()*, *kill()*, *pthread_kill()*, or
43258 *sigqueue()*, the behavior is undefined if the signal handler refers to any object with static storage
43259 duration other than by assigning a value to an object declared as volatile **sig_atomic_t**, or if the
43260 signal handler calls any function in the standard library other than one of the functions listed in
43261 Section 2.4 (on page 28). Furthermore, if such a call fails, the value of *errno* is unspecified.

43262 At program start-up, the equivalent of:

43263 `signal(sig, SIG_IGN);`

43264 is executed for some signals, and the equivalent of:

43265 `signal(sig, SIG_DFL);`

43266 CX is executed for all other signals (see *exec*).

43267 RETURN VALUE

43268 If the request can be honored, *signal()* shall return the value of *func* for the most recent call to
43269 *signal()* for the specified signal *sig*. Otherwise, SIG_ERR shall be returned and a positive value
43270 shall be stored in *errno*.

43271 ERRORS

43272 The *signal()* function shall fail if:

43273 CX [EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

43275 The *signal()* function may fail if:

43276 CX [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be caught or ignored (or both).

43278 EXAMPLES

43279 None.

43280 APPLICATION USAGE

43281 The *sigaction()* function provides a more comprehensive and reliable mechanism for controlling signals; new applications should use *sigaction()* rather than *signal()*.

43283 RATIONALE

43284 None.

43285 FUTURE DIRECTIONS

43286 None.

43287 SEE ALSO

43288 Section 2.4 (on page 28), *exec*, *pause()*, *sigaction()*, *sigsuspend()*, *waitid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>

43290 CHANGE HISTORY

43291 First released in Issue 1. Derived from Issue 1 of the SVID.

43292 Issue 5

43293 Moved from X/OPEN UNIX extension to BASE.

43294 The DESCRIPTION is updated to indicate that the *sigpause()* function restores the process' signal mask to its original state before returning.

43296 The RETURN VALUE section is updated to indicate that the *sigpause()* function suspends execution of the process until a signal is received, whereupon it returns -1 and sets *errno* to [EINTR].

43299 Issue 6

43300 Extensions beyond the ISO C standard are marked.

43301 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

43302 The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

43303 References to the *wait3()* function are removed.

43304 The *sighold()*, *sigignore()*, *sigrelse()*, and *sigset()* functions are split out onto their own reference page.

43306 NAME

43307 signbit — test sign

43308 SYNOPSIS

```
43309 #include <math.h>
43310 int signbit(real-floating x);
```

43311 DESCRIPTION

43312 CX The functionality described on this reference page is aligned with the ISO C standard. Any
43313 conflict between the requirements described here and the ISO C standard is unintentional. This
43314 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

43315 The *signbit()* macro shall determine whether the sign of its argument value is negative. NaNs,
43316 zeros, and infinities have a sign bit.

43317 RETURN VALUE

43318 The *signbit()* macro shall return a non-zero value if and only if the sign of its argument value is
43319 negative.

43320 ERRORS

43321 No errors are defined.

43322 EXAMPLES

43323 None.

43324 APPLICATION USAGE

43325 None.

43326 RATIONALE

43327 None.

43328 FUTURE DIRECTIONS

43329 None.

43330 SEE ALSO

43331 *fpclassify()*, *isfinite()*, *isinf()*, *isnan()*, *isnormal()*, the Base Definitions volume of
43332 IEEE Std 1003.1-2001, <**math.h**>

43333 CHANGE HISTORY

43334 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

43335 NAME

43336 `sigpause` — remove a signal from the signal mask and suspend the thread

43337 SYNOPSIS

43338 XSI `#include <signal.h>`

43339 `int sigpause(int sig);`

43340

43341 DESCRIPTION

43342 Refer to *sighold()*.

43343 NAME

43344 *sigpending* — examine pending signals

43345 SYNOPSIS

43346 CX

```
#include <signal.h>
```

43347

```
int sigpending(sigset_t *set);
```

43348

43349 DESCRIPTION

43350 The *sigpending*() function shall store, in the location referenced by the *set* argument, the set of signals that are blocked from delivery to the calling thread and that are pending on the process or the calling thread.

43353 RETURN VALUE

43354 Upon successful completion, *sigpending*() shall return 0; otherwise, -1 shall be returned and *errno* set to indicate the error.

43356 ERRORS

43357 No errors are defined.

43358 EXAMPLES

43359 None.

43360 APPLICATION USAGE

43361 None.

43362 RATIONALE

43363 None.

43364 FUTURE DIRECTIONS

43365 None.

43366 SEE ALSO

43367 *sigaddset()*, *sigdelset()*, *sigemptyset()*, *sigfillset()*, *sigismember()*, *sigprocmask()*, the Base Definitions volume of IEEE Std 1003.1-2001, <*signal.h*>

43369 CHANGE HISTORY

43370 First released in Issue 3.

43371 Issue 5

43372 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

43373 Issue 6

43374 The SYNOPSIS is marked CX since the presence of this function in the <*signal.h*> header is an extension over the ISO C standard.

43376 NAME

43377 sigprocmask — examine and change blocked signals

43378 SYNOPSIS

43379 CX #include <signal.h>
43380 int sigprocmask(int *how*, const sigset_t *restrict *set*,
43381 sigset_t *restrict *oset*);
43382

43383 DESCRIPTION

43384 Refer to *pthread_sigmask()*.

43385 NAME

43386 sigqueue — queue a signal to a process (**REALTIME**)

43387 SYNOPSIS

43388 RTS #include <signal.h>

43389 int sigqueue(pid_t pid, int signo, const union sigval value);

43390

43391 DESCRIPTION

43392 The *sigqueue()* function shall cause the signal specified by *signo* to be sent with the value
43393 specified by *value* to the process specified by *pid*. If *signo* is zero (the null signal), error checking
43394 is performed but no signal is actually sent. The null signal can be used to check the validity of
43395 *pid*.

43396 The conditions required for a process to have permission to queue a signal to another process
43397 are the same as for the *kill()* function.

43398 The *sigqueue()* function shall return immediately. If SA_SIGINFO is set for *signo* and if the
43399 resources were available to queue the signal, the signal shall be queued and sent to the receiving
43400 process. If SA_SIGINFO is not set for *signo*, then *signo* shall be sent at least once to the receiving
43401 process; it is unspecified whether *value* shall be sent to the receiving process as a result of this
43402 call.

43403 If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked
43404 for the calling thread and if no other thread has *signo* unblocked or is waiting in a *sigwait()*
43405 function for *signo*, either *signo* or at least the pending, unblocked signal shall be delivered to the
43406 calling thread before the *sigqueue()* function returns. Should any multiple pending signals in the
43407 range SIGRTMIN to SIGRTMAX be selected for delivery, it shall be the lowest numbered one.
43408 The selection order between realtime and non-realtime signals, or between multiple pending
43409 non-realtime signals, is unspecified.

43410 RETURN VALUE

43411 Upon successful completion, the specified signal shall have been queued, and the *sigqueue()*
43412 function shall return a value of zero. Otherwise, the function shall return a value of -1 and set
43413 *errno* to indicate the error.

43414 ERRORS

43415 The *sigqueue()* function shall fail if:

43416 [EAGAIN] No resources are available to queue the signal. The process has already
43417 queued {SIGQUEUE_MAX} signals that are still pending at the receiver(s), or
43418 a system-wide resource limit has been exceeded.

43419 [EINVAL] The value of the *signo* argument is an invalid or unsupported signal number.

43420 [EPERM] The process does not have the appropriate privilege to send the signal to the
43421 receiving process.

43422 [ESRCH] The process *pid* does not exist.

43423 EXAMPLES

43424 None.

43425 APPLICATION USAGE

43426 None.

43427 RATIONALE

43428 The *sigqueue()* function allows an application to queue a realtime signal to itself or to another
43429 process, specifying the application-defined value. This is common practice in realtime
43430 applications on existing realtime systems. It was felt that specifying another function in the
43431 *sig...* name space already carved out for signals was preferable to extending the interface to
43432 *kill()*.

43433 Such a function became necessary when the put/get event function of the message queues was
43434 removed. It should be noted that the *sigqueue()* function implies reduced performance in a
43435 security-conscious implementation as the access permissions between the sender and receiver
43436 have to be checked on each send when the *pid* is resolved into a target process. Such access
43437 checks were necessary only at message queue open in the previous interface.

43438 The standard developers required that *sigqueue()* have the same semantics with respect to the
43439 null signal as *kill()*, and that the same permission checking be used. But because of the difficulty
43440 of implementing the “broadcast” semantic of *kill()* (for example, to process groups) and the
43441 interaction with resource allocation, this semantic was not adopted. The *sigqueue()* function
43442 queues a signal to a single process specified by the *pid* argument.

43443 The *sigqueue()* function can fail if the system has insufficient resources to queue the signal. An
43444 explicit limit on the number of queued signals that a process could send was introduced. While
43445 the limit is “per-sender”, this volume of IEEE Std 1003.1-2001 does not specify that the resources
43446 be part of the state of the sender. This would require either that the sender be maintained after
43447 exit until all signals that it had sent to other processes were handled or that all such signals that
43448 had not yet been acted upon be removed from the queue(s) of the receivers. This volume of
43449 IEEE Std 1003.1-2001 does not preclude this behavior, but an implementation that allocated
43450 queuing resources from a system-wide pool (with per-sender limits) and that leaves queued
43451 signals pending after the sender exits is also permitted.

43452 FUTURE DIRECTIONS

43453 None.

43454 SEE ALSO

43455 Section 2.8.1 (on page 41), the Base Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>

43456 CHANGE HISTORY

43457 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
43458 POSIX Threads Extension.

43459 Issue 6

43460 The *sigqueue()* function is marked as part of the Realtime Signals Extension option.

43461 The [ENOSYS] error condition has been removed as stubs need not be provided if an
43462 implementation does not support the Realtime Signals Extension option.

43463 **NAME**43464 **sigrelse, sigset — signal management**43465 **SYNOPSIS**

43466 XSI #include <signal.h>

43467 int sigrelse(int *sig*);43468 void (*sigset(int *sig*, void (**disp*)(int)))(int);

43469

43470 **DESCRIPTION**43471 Refer to *sighold*().

43472 NAME

43473 `sigsetjmp` — set jump point for a non-local goto

43474 SYNOPSIS

43475 CX `#include <setjmp.h>`

43476 `int sigsetjmp(sigjmp_buf env, int savemask);`

43477

43478 DESCRIPTION

43479 The `sigsetjmp()` function shall be equivalent to the `setjmp()` function, except as follows:

- 43480 • References to `setjmp()` are equivalent to `sigsetjmp()`.
- 43481 • References to `longjmp()` are equivalent to `siglongjmp()`.
- 43482 • If the value of the `savemask` argument is not 0, `sigsetjmp()` shall also save the current signal
43483 mask of the calling thread as part of the calling environment.

43484 RETURN VALUE

43485 If the return is from a successful direct invocation, `sigsetjmp()` shall return 0. If the return is from
43486 a call to `siglongjmp()`, `sigsetjmp()` shall return a non-zero value.

43487 ERRORS

43488 No errors are defined.

43489 EXAMPLES

43490 None.

43491 APPLICATION USAGE

43492 The distinction between `setjmp()`/`longjmp()` and `sigsetjmp()`/`siglongjmp()` is only significant for
43493 programs which use `sigaction()`, `sigprocmask()`, or `sigsuspend()`.

43494 Note that since this function is defined in terms of `setjmp()`, if `savemask` is zero, it is unspecified
43495 whether the signal mask is saved.

43496 RATIONALE

43497 The ISO C standard specifies various restrictions on the usage of the `setjmp()` macro in order to
43498 permit implementors to recognize the name in the compiler and not implement an actual
43499 function. These same restrictions apply to the `sigsetjmp()` macro.

43500 There are processors that cannot easily support these calls, but this was not considered a
43501 sufficient reason to exclude them.

43502 4.2 BSD, 4.3 BSD, and XSI-conformant systems provide functions named `_setjmp()` and
43503 `_longjmp()` that, together with `setjmp()` and `longjmp()`, provide the same functionality as
43504 `sigsetjmp()` and `siglongjmp()`. On those systems, `setjmp()` and `longjmp()` save and restore signal
43505 masks, while `_setjmp()` and `_longjmp()` do not. On System V Release 3 and in corresponding
43506 issues of the SVID, `setjmp()` and `longjmp()` are explicitly defined not to save and restore signal
43507 masks. In order to permit existing practice in both cases, the relation of `setjmp()` and `longjmp()` to
43508 signal masks is not specified, and a new set of functions is defined instead.

43509 The `longjmp()` and `siglongjmp()` functions operate as in the previous issue provided the matching
43510 `setjmp()` or `sigsetjmp()` has been performed in the same thread. Non-local jumps into contexts
43511 saved by other threads would be at best a questionable practice and were not considered worthy
43512 of standardization.

43513 FUTURE DIRECTIONS

43514 None.

43515 SEE ALSO

43516 *siglongjmp()*, *signal()*, *sigprocmask()*, *sigsuspend()*, the Base Definitions volume of
43517 IEEE Std 1003.1-2001, <**setjmp.h**>

43518 CHANGE HISTORY

43519 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

43520 Issue 5

43521 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

43522 Issue 6

43523 The DESCRIPTION is reworded in terms of *setjmp()*.

43524 The SYNOPSIS is marked CX since the presence of this function in the <**setjmp.h**> header is an
43525 extension over the ISO C standard.

43526 NAME

43527 *sigsuspend* — wait for a signal

43528 SYNOPSIS

43529 CX `#include <signal.h>`

43530 `int sigsuspend(const sigset_t *sigmask);`

43531

43532 DESCRIPTION

43533 The *sigsuspend*() function shall replace the current signal mask of the calling thread with the set
43534 of signals pointed to by *sigmask* and then suspend the thread until delivery of a signal whose
43535 action is either to execute a signal-catching function or to terminate the process. This shall not
43536 cause any other signals that may have been pending on the process to become pending on the
43537 thread.

43538 If the action is to terminate the process then *sigsuspend*() shall never return. If the action is to
43539 execute a signal-catching function, then *sigsuspend*() shall return after the signal-catching
43540 function returns, with the signal mask restored to the set that existed prior to the *sigsuspend*()
43541 call.

43542 It is not possible to block signals that cannot be ignored. This is enforced by the system without
43543 causing an error to be indicated.

43544 RETURN VALUE

43545 Since *sigsuspend*() suspends thread execution indefinitely, there is no successful completion
43546 return value. If a return occurs, -1 shall be returned and *errno* set to indicate the error.

43547 ERRORS

43548 The *sigsuspend*() function shall fail if:

43549 [EINTR] A signal is caught by the calling process and control is returned from the
43550 signal-catching function.

43551 EXAMPLES

43552 None.

43553 APPLICATION USAGE

43554 Normally, at the beginning of a critical code section, a specified set of signals is blocked using
43555 the *sigprocmask*() function. When the thread has completed the critical section and needs to wait
43556 for the previously blocked signal(s), it pauses by calling *sigsuspend*() with the mask that was
43557 returned by the *sigprocmask*() call.

43558 RATIONALE

43559 None.

43560 FUTURE DIRECTIONS

43561 None.

43562 SEE ALSO

43563 Section 2.4 (on page 28), *pause*(), *sigaction*(), *sigaddset*(), *sigdelset*(), *sigemptyset*(), *sigfillset*(), the
43564 Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`

43565 CHANGE HISTORY

43566 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

43567 Issue 5

43568 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

43569 Issue 6

43570 The text in the RETURN VALUE section has been changed from “suspends process execution”
43571 to “suspends thread execution”. This reflects IEEE PASC Interpretation 1003.1c #40.

43572 Text in the APPLICATION USAGE section has been replaced.

43573 The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an
43574 extension over the ISO C standard.

43575 NAME

43576 sigtimedwait, sigwaitinfo — wait for queued signals (**REALTIME**)

43577 SYNOPSIS

```
43578 RTS #include <signal.h>
43579     int sigtimedwait(const sigset_t *restrict set,
43580             siginfo_t *restrict info,
43581             const struct timespec *restrict timeout);
43582     int sigwaitinfo(const sigset_t *restrict set,
43583             siginfo_t *restrict info);
```

43585 DESCRIPTION

43586 The *sigtimedwait()* function shall be equivalent to *sigwaitinfo()* except that if none of the signals
 43587 specified by *set* are pending, *sigtimedwait()* shall wait for the time interval specified in the
 43588 **timespec** structure referenced by *timeout*. If the **timespec** structure pointed to by *timeout* is
 43589 zero-valued and if none of the signals specified by *set* are pending, then *sigtimedwait()* shall
 43590 MON return immediately with an error. If *timeout* is the NULL pointer, the behavior is unspecified. If
 43591 the Monotonic Clock option is supported, the **CLOCK_MONOTONIC** clock shall be used to
 43592 measure the time interval specified by the *timeout* argument.

43593 The *sigwaitinfo()* function selects the pending signal from the set specified by *set*. Should any of
 43594 multiple pending signals in the range **SIGRTMIN** to **SIGRTMAX** be selected, it shall be the
 43595 lowest numbered one. The selection order between realtime and non-realtime signals, or
 43596 between multiple pending non-realtime signals, is unspecified. If no signal in *set* is pending at
 43597 the time of the call, the calling thread shall be suspended until one or more signals in *set* become
 43598 pending or until it is interrupted by an unblocked, caught signal.

43599 The *sigwaitinfo()* function shall be equivalent to the *sigwait()* function if the *info* argument is
 43600 NULL. If the *info* argument is non-NULL, the *sigwaitinfo()* function shall be equivalent to
 43601 *sigwait()*, except that the selected signal number shall be stored in the *si_signo* member, and the
 43602 cause of the signal shall be stored in the *si_code* member. If any value is queued to the selected
 43603 signal, the first such queued value shall be dequeued and, if the *info* argument is non-NULL, the
 43604 value shall be stored in the *si_value* member of *info*. The system resource used to queue the
 43605 signal shall be released and returned to the system for other use. If no value is queued, the
 43606 content of the *si_value* member is undefined. If no further signals are queued for the selected
 43607 signal, the pending indication for that signal shall be reset.

43608 RETURN VALUE

43609 Upon successful completion (that is, one of the signals specified by *set* is pending or is
 43610 generated) *sigwaitinfo()* and *sigtimedwait()* shall return the selected signal number. Otherwise,
 43611 the function shall return a value of -1 and set *errno* to indicate the error.

43612 ERRORS

43613 The *sigtimedwait()* function shall fail if:

43614 [EAGAIN] No signal specified by *set* was generated within the specified timeout period.

43615 The *sigtimedwait()* and *sigwaitinfo()* functions may fail if:

43616 [EINTR] The wait was interrupted by an unblocked, caught signal. It shall be
 43617 documented in system documentation whether this error causes these
 43618 functions to fail.

43619 The *sigtimedwait()* function may also fail if:
43620 [EINVAL] The *timeout* argument specified a *tv_nsec* value less than zero or greater than
43621 or equal to 1 000 million.
43622 An implementation should only check for this error if no signal is pending in *set* and it is 2
43623 necessary to wait.

43624 EXAMPLES

43625 None.

43626 APPLICATION USAGE

43627 The *sigtimedwait()* function times out and returns an [EAGAIN] error. Application writers
43628 should note that this is inconsistent with other functions such as *pthread_cond_timedwait()* that
43629 return [ETIMEDOUT].

43630 RATIONALE

43631 Existing programming practice on realtime systems uses the ability to pause waiting for a
43632 selected set of events and handle the first event that occurs in-line instead of in a signal-handling
43633 function. This allows applications to be written in an event-directed style similar to a state
43634 machine. This style of programming is useful for largescale transaction processing in which the
43635 overall throughput of an application and the ability to clearly track states are more important
43636 than the ability to minimize the response time of individual event handling.

43637 It is possible to construct a signal-waiting macro function out of the realtime signal function
43638 mechanism defined in this volume of IEEE Std 1003.1-2001. However, such a macro has to
43639 include the definition of a generalized handler for all signals to be waited on. A significant
43640 portion of the overhead of handler processing can be avoided if the signal-waiting function is
43641 provided by the kernel. This volume of IEEE Std 1003.1-2001 therefore provides two signal-
43642 waiting functions—one that waits indefinitely and one with a timeout—as part of the overall
43643 realtime signal function specification.

43644 The specification of a function with a timeout allows an application to be written that can be
43645 broken out of a wait after a set period of time if no event has occurred. It was argued that setting
43646 a timer event before the wait and recognizing the timer event in the wait would also implement
43647 the same functionality, but at a lower performance level. Because of the performance
43648 degradation associated with the user-level specification of a timer event and the subsequent
43649 cancellation of that timer event after the wait completes for a valid event, and the complexity
43650 associated with handling potential race conditions associated with the user-level method, the
43651 separate function has been included.

43652 Note that the semantics of the *sigwaitinfo()* function are nearly identical to that of the *sigwait()*
43653 function defined by this volume of IEEE Std 1003.1-2001. The only difference is that *sigwaitinfo()*
43654 returns the queued signal value in the *value* argument. The return of the queued value is
43655 required so that applications can differentiate between multiple events queued to the same
43656 signal number.

43657 The two distinct functions are being maintained because some implementations may choose to
43658 implement the POSIX Threads Extension functions and not implement the queued signals
43659 extensions. Note, though, that *sigwaitinfo()* does not return the queued value if the *value*
43660 argument is NULL, so the POSIX Threads Extension *sigwait()* function can be implemented as a
43661 macro on *sigwaitinfo()*.

43662 The *sigtimedwait()* function was separated from the *sigwaitinfo()* function to address concerns
43663 regarding the overloading of the *timeout* pointer to indicate indefinite wait (no timeout), timed
43664 wait, and immediate return, and concerns regarding consistency with other functions where the
43665 conditional and timed waits were separate functions from the pure blocking function. The

43666 semantics of *sigtimedwait()* are specified such that *sigwaitinfo()* could be implemented as a
43667 macro with a NULL pointer for *timeout*.

43668 The *sigwait* functions provide a synchronous mechanism for threads to wait for
43669 asynchronously-generated signals. One important question was how many threads that are
4370 suspended in a call to a *sigwait()* function for a signal should return from the call when the
4371 signal is sent. Four choices were considered:

- 43672 1. Return an error for multiple simultaneous calls to *sigwait* functions for the same signal.
- 43673 2. One or more threads return.
- 43674 3. All waiting threads return.
- 43675 4. Exactly one thread returns.

43676 Prohibiting multiple calls to *sigwait()* for the same signal was felt to be overly restrictive. The
43677 “one or more” behavior made implementation of conforming packages easy at the expense of
43678 forcing POSIX threads clients to protect against multiple simultaneous calls to *sigwait()* in
43679 application code in order to achieve predictable behavior. There was concern that the “all
43680 waiting threads” behavior would result in “signal broadcast storms”, consuming excessive CPU
43681 resources by replicating the signals in the general case. Furthermore, no convincing examples
43682 could be presented that delivery to all was either simpler or more powerful than delivery to one.

43683 Thus, the consensus was that exactly one thread that was suspended in a call to a *sigwait*
43684 function for a signal should return when that signal occurs. This is not an onerous restriction as:

- 43685 • A multi-way signal wait can be built from the single-way wait.
- 43686 • Signals should only be handled by application-level code, as library routines cannot guess
43687 what the application wants to do with signals generated for the entire process.
- 43688 • Applications can thus arrange for a single thread to wait for any given signal and call any
43689 needed routines upon its arrival.

43690 In an application that is using signals for interprocess communication, signal processing is
43691 typically done in one place. Alternatively, if the signal is being caught so that process cleanup
43692 can be done, the signal handler thread can call separate process cleanup routines for each
43693 portion of the application. Since the application main line started each portion of the application,
43694 it is at the right abstraction level to tell each portion of the application to clean up.

43695 Certainly, there exist programming styles where it is logical to consider waiting for a single
43696 signal in multiple threads. A simple *sigwait_multiple()* routine can be constructed to achieve this
43697 goal. A possible implementation would be to have each *sigwait_multiple()* caller registered as
43698 having expressed interest in a set of signals. The caller then waits on a thread-specific condition
43699 variable. A single server thread calls a *sigwait()* function on the union of all registered signals.
43700 When the *sigwait()* function returns, the appropriate state is set and condition variables are
43701 broadcast. New *sigwait_multiple()* callers may cause the pending *sigwait()* call to be canceled
43702 and reissued in order to update the set of signals being waited for.

43703 FUTURE DIRECTIONS

43704 None.

43705 SEE ALSO

43706 Section 2.8.1 (on page 41), *pause()*, *pthread_sigmask()*, *sigaction()*, *sigpending()*, *sigsuspend()*,
43707 *sigwait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>, <time.h>

43708 CHANGE HISTORY

43709 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
43710 POSIX Threads Extension.

43711 Issue 6

43712 These functions are marked as part of the Realtime Signals Extension option.

43713 The Open Group Corrigendum U035/3 is applied. The SYNOPSIS of the *sigwaitinfo()* function
43714 has been corrected so that the second argument is of type **siginfo_t** *.

43715 The [ENOSYS] error condition has been removed as stubs need not be provided if an
43716 implementation does not support the Realtime Signals Extension option.

43717 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the
43718 CLOCK_MONOTONIC clock, if supported, is used to measure timeout intervals.

43719 The **restrict** keyword is added to the *sigtimedwait()* and *sigwaitinfo()* prototypes for alignment
43720 with the ISO/IEC 9899:1999 standard.

43721 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/130 is applied, restoring wording in the 2
43722 RETURN VALUE section to that in the original base document (“An implementation should 2
43723 only check for this error if no signal is pending in *set* and it is necessary to wait”). 2

43724 NAME

43725 *sigwait* — wait for queued signals

43726 SYNOPSIS

43727 CX

```
#include <signal.h>
```

43728

```
int sigwait(const sigset_t *restrict set, int *restrict sig);
```

43729

43730 DESCRIPTION

43731 The *sigwait()* function shall select a pending signal from *set*, atomically clear it from the system's
 43732 set of pending signals, and return that signal number in the location referenced by *sig*. If prior to
 43733 the call to *sigwait()* there are multiple pending instances of a single signal number, it is
 43734 implementation-defined whether upon successful return there are any remaining pending
 43735 RTS signals for that signal number. If the implementation supports queued signals and there are
 43736 multiple signals queued for the signal number selected, the first such queued signal shall cause a
 43737 return from *sigwait()* and the remainder shall remain queued. If no signal in *set* is pending at the
 43738 time of the call, the thread shall be suspended until one or more becomes pending. The signals
 43739 defined by *set* shall have been blocked at the time of the call to *sigwait()*; otherwise, the behavior
 43740 is undefined. The effect of *sigwait()* on the signal actions for the signals in *set* is unspecified.

43741 If more than one thread is using *sigwait()* to wait for the same signal, no more than one of these
 43742 threads shall return from *sigwait()* with the signal number. If more than a single thread is
 43743 blocked in *sigwait()* for a signal when that signal is generated for the process, it is unspecified
 43744 which of the waiting threads returns from *sigwait()*. If the signal is generated for a specific
 43745 thread, as by *pthread_kill()*, only that thread shall return. 2
 2
 2
 2

43746 RTS Should any of the multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it
 43747 shall be the lowest numbered one. The selection order between realtime and non-realtime
 43748 signals, or between multiple pending non-realtime signals, is unspecified. 2

43749 RETURN VALUE

43750 Upon successful completion, *sigwait()* shall store the signal number of the received signal at the
 43751 location referenced by *sig* and return zero. Otherwise, an error number shall be returned to
 43752 indicate the error.

43753 ERRORS

43754 The *sigwait()* function may fail if:

43755 [EINVAL] The *set* argument contains an invalid or unsupported signal number.

43756 EXAMPLES

43757 None.

43758 APPLICATION USAGE

43759 None.

43760 RATIONALE

43761 To provide a convenient way for a thread to wait for a signal, this volume of
 43762 IEEE Std 1003.1-2001 provides the *sigwait()* function. For most cases where a thread has to wait
 43763 for a signal, the *sigwait()* function should be quite convenient, efficient, and adequate.

43764 However, requests were made for a lower-level primitive than *sigwait()* and for semaphores that
 43765 could be used by threads. After some consideration, threads were allowed to use semaphores
 43766 and *sem_post()* was defined to be async-signal and async-cancel-safe.

43767 In summary, when it is necessary for code run in response to an asynchronous signal to notify a
 43768 thread, *sigwait()* should be used to handle the signal. Alternatively, if the implementation
 43769 provides semaphores, they also can be used, either following *sigwait()* or from within a signal

43770 handling routine previously registered with *sigaction()*.

43771 **FUTURE DIRECTIONS**

43772 None.

43773 **SEE ALSO**

43774 Section 2.4 (on page 28), Section 2.8.1 (on page 41), *pause()*, *pthread_sigmask()*, *sigaction()*,
43775 *sigpending()*, *sigsuspend()*, *sigwaitinfo()*, the Base Definitions volume of IEEE Std 1003.1-2001,
43776 <**signal.h**>, <**time.h**>

43777 **CHANGE HISTORY**

43778 First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
43779 POSIX Threads Extension.

43780 **Issue 6**

43781 The **restrict** keyword is added to the *sigwait()* prototype for alignment with the
43782 ISO/IEC 9899:1999 standard.

43783 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/131 is applied, updating the 2
43784 DESCRIPTION section to state that if more than a single thread is blocked in *sigwait()*, it is 2
43785 unspecified which of the waiting threads returns, and that if a signal is generated for a specific 2
43786 thread only that thread shall return. 2

43787 NAME

43788 **sigwaitinfo** — wait for queued signals (**REALTIME**)

43789 SYNOPSIS

43790 RTS `#include <signal.h>`

43791 `int sigwaitinfo(const sigset_t *restrict set, siginfo_t *restrict info);`

43792

43793 DESCRIPTION

43794 Refer to *sigtimedwait()*.

43795 NAME

43796 sin, sinf, sinl — sine function

43797 SYNOPSIS

```
43798     #include <math.h>
43799     double sin(double x);
43800     float sinf(float x);
43801     long double sinl(long double x);
```

43802 DESCRIPTION

43803 CX 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.

43806 These functions shall compute the sine of their argument x , measured in radians.

43807 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.

43811 RETURN VALUE

43812 Upon successful completion, these functions shall return the sine of x .

43813 MX If x is NaN, a NaN shall be returned.

43814 If x is ± 0 , x shall be returned.

43815 If x is subnormal, a range error may occur and x should be returned.

43816 If x is $\pm\infty$, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

43818 ERRORS

43819 These functions shall fail if:

43820 MX Domain Error The x argument is $\pm\infty$.

43821 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.

43825 These functions may fail if:

43826 MX Range Error The value of x is subnormal

43827 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 underflow floating-point exception shall be raised.

43831 EXAMPLES**43832 Taking the Sine of a 45-Degree Angle**

```
43833 #include <math.h>
43834 ...
43835 double radians = 45.0 * M_PI / 180;
43836 double result;
43837 ...
43838 result = sin(radians);
```

43839 APPLICATION USAGE

43840 These functions may lose accuracy when their argument is near a multiple of π or is far from 0.0.

43841 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
43842 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

43843 RATIONALE

43844 None.

43845 FUTURE DIRECTIONS

43846 None.

43847 SEE ALSO

43848 *asin()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
43849 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

43850 CHANGE HISTORY

43851 First released in Issue 1. Derived from Issue 1 of the SVID.

43852 Issue 5

43853 The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes
43854 in previous issues.

43855 Issue 6

43856 The *sinf()* and *sinl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

43857 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
43858 revised to align with the ISO/IEC 9899:1999 standard.

43859 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
43860 marked.

43861 NAME

43862 `sinh, sinhf, sinhl — hyperbolic sine functions`

43863 SYNOPSIS

```
43864     #include <math.h>
43865
43866     double sinh(double x);
43867     float sinhf(float x);
43868     long double sinhl(long double x);
```

43868 DESCRIPTION

43869 CX 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.

43872 These functions shall compute the hyperbolic sine of their argument *x*.

43873 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.

43877 RETURN VALUE

43878 Upon successful completion, these functions shall return the hyperbolic sine of *x*.

43879 If the result would cause an overflow, a range error shall occur and $\pm\text{HUGE_VAL}$,
 43880 $\pm\text{HUGE_VALF}$, and $\pm\text{HUGE_VALL}$ (with the same sign as *x*) shall be returned as appropriate for
 43881 the type of the function.

43882 MX If *x* is NaN, a NaN shall be returned.

43883 If *x* is ± 0 or $\pm\text{Inf}$, *x* shall be returned.

43884 If *x* is subnormal, a range error may occur and *x* should be returned.

43885 ERRORS

43886 These functions shall fail if:

43887 Range Error The result would cause an overflow.

43888 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 43889 then *errno* shall be set to [ERANGE]. If the integer expression
 43890 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow
 43891 floating-point exception shall be raised.

43892 These functions may fail if:

43893 MX Range Error The value *x* is subnormal.

43894 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
 43895 then *errno* shall be set to [ERANGE]. If the integer expression
 43896 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the underflow
 43897 floating-point exception shall be raised.

43898 EXAMPLES

43899 None.

43900 APPLICATION USAGE

43901 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
43902 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

43903 RATIONALE

43904 None.

43905 FUTURE DIRECTIONS

43906 None.

43907 SEE ALSO

43908 *asinh()*, *cosh()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *tanh()*, the Base Definitions volume of
43909 IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
43910 *<math.h>*

43911 CHANGE HISTORY

43912 First released in Issue 1. Derived from Issue 1 of the SVID.

43913 Issue 5

43914 The DESCRIPTION is updated to indicate how an application should check for an error. This
43915 text was previously published in the APPLICATION USAGE section.

43916 Issue 6

43917 The *sinhf()* and *sinhl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.
43918 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
43919 revised to align with the ISO/IEC 9899:1999 standard.
43920 IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
43921 marked.

43922 **NAME**43923 **sinl** — sine function43924 **SYNOPSIS**

43925 #include <math.h>

43926 long double sinl(long double x);

43927 **DESCRIPTION**43928 Refer to *sin()*.

43929 **NAME**

43930 sleep — suspend execution for an interval of time

43931 **SYNOPSIS**

43932 #include <unistd.h>
43933 unsigned sleep(unsigned seconds);

43934 **DESCRIPTION**

43935 The *sleep()* function shall cause the calling thread to be suspended from execution until either
43936 the number of realtime seconds specified by the argument *seconds* has elapsed or a signal is
43937 delivered to the calling thread and its action is to invoke a signal-catching function or to
43938 terminate the process. The suspension time may be longer than requested due to the scheduling
43939 of other activity by the system.

43940 If a SIGALRM signal is generated for the calling process during execution of *sleep()* and if the
43941 SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *sleep()*
43942 returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also
43943 unspecified whether it remains pending after *sleep()* returns or it is discarded.

43944 If a SIGALRM signal is generated for the calling process during execution of *sleep()*, except as a
43945 result of a prior call to *alarm()*, and if the SIGALRM signal is not being ignored or blocked from
43946 delivery, it is unspecified whether that signal has any effect other than causing *sleep()* to return.

43947 If a signal-catching function interrupts *sleep()* and examines or changes either the time a
43948 SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or
43949 whether the SIGALRM signal is blocked from delivery, the results are unspecified.

43950 If a signal-catching function interrupts *sleep()* and calls *siglongjmp()* or *longjmp()* to restore an
43951 environment saved prior to the *sleep()* call, the action associated with the SIGALRM signal and
43952 the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also
43953 unspecified whether the SIGALRM signal is blocked, unless the process' signal mask is restored
43954 as part of the environment.

43955 XSI Interactions between *sleep()* and any of *setitimer()*, *ualarm()*, or *usleep()* are unspecified.

43956 **RETURN VALUE**

43957 If *sleep()* returns because the requested time has elapsed, the value returned shall be 0. If *sleep()*
43958 returns due to delivery of a signal, the return value shall be the “unslept” amount (the requested
43959 time minus the time actually slept) in seconds.

43960 **ERRORS**

43961 No errors are defined.

43962 **EXAMPLES**

43963 None.

43964 **APPLICATION USAGE**

43965 None.

43966 **RATIONALE**

43967 There are two general approaches to the implementation of the *sleep()* function. One is to use the
43968 *alarm()* function to schedule a SIGALRM signal and then suspend the calling thread waiting for
43969 that signal. The other is to implement an independent facility. This volume of
43970 IEEE Std 1003.1-2001 permits either approach.

43971 In order to comply with the requirement that no primitive shall change a process attribute unless
43972 explicitly described by this volume of IEEE Std 1003.1-2001, an implementation using SIGALRM
43973 must carefully take into account any SIGALRM signal scheduled by previous *alarm()* calls, the

43974 action previously established for SIGALRM, and whether SIGALRM was blocked. If a SIGALRM
43975 has been scheduled before the *sleep()* would ordinarily complete, the *sleep()* must be shortened
43976 to that time and a SIGALRM generated (possibly simulated by direct invocation of the signal-
43977 catching function) before *sleep()* returns. If a SIGALRM has been scheduled after the *sleep()*
43978 would ordinarily complete, it must be rescheduled for the same time before *sleep()* returns. The
43979 action and blocking for SIGALRM must be saved and restored.

43980 Historical implementations often implement the SIGALRM-based version using *alarm()* and
43981 *pause()*. One such implementation is prone to infinite hangups, as described in *pause()*. Another
43982 such implementation uses the C-language *setjmp()* and *longjmp()* functions to avoid that
43983 window. That implementation introduces a different problem: when the SIGALRM signal
43984 interrupts a signal-catching function installed by the user to catch a different signal, the
43985 *longjmp()* aborts that signal-catching function. An implementation based on *sigprocmask()*,
43986 *alarm()*, and *sigsuspend()* can avoid these problems.

43987 Despite all reasonable care, there are several very subtle, but detectable and unavoidable,
43988 differences between the two types of implementations. These are the cases mentioned in this
43989 volume of IEEE Std 1003.1-2001 where some other activity relating to SIGALRM takes place, and
43990 the results are stated to be unspecified. All of these cases are sufficiently unusual as not to be of
43991 concern to most applications.

43992 See also the discussion of the term *realtime* in *alarm()*.

43993 Since *sleep()* can be implemented using *alarm()*, the discussion about alarms occurring early
43994 under *alarm()* applies to *sleep()* as well.

43995 Application writers should note that the type of the argument *seconds* and the return value of
43996 *sleep()* is **unsigned**. That means that a Strictly Conforming POSIX System Interfaces Application
43997 cannot pass a value greater than the minimum guaranteed value for {UINT_MAX}, which the
43998 ISO C standard sets as 65 535, and any application passing a larger value is restricting its
43999 portability. A different type was considered, but historical implementations, including those
44000 with a 16-bit **int** type, consistently use either **unsigned** or **int**.

44001 Scheduling delays may cause the process to return from the *sleep()* function significantly after
44002 the requested time. In such cases, the return value should be set to zero, since the formula
44003 (requested time minus the time actually spent) yields a negative number and *sleep()* returns an
44004 **unsigned**.

44005 FUTURE DIRECTIONS

44006 None.

44007 SEE ALSO

44008 *alarm()*, *getitimer()*, *nanosleep()*, *pause()*, *sigaction()*, *sigsetjmp()*, *ualarm()*, *usleep()*, the Base
44009 Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

44010 CHANGE HISTORY

44011 First released in Issue 1. Derived from Issue 1 of the SVID.

44012 Issue 5

44013 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

44014 Issue 6

44015 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/132 is applied, making a correction in the
44016 RATIONALE section. 2

44017 **NAME**

44018 snprintf — print formatted output

44019 **SYNOPSIS**

44020 #include <stdio.h>

44021 int snprintf(char *restrict *s*, size_t *n*,
44022 const char *restrict *format*, ...);44023 **DESCRIPTION**44024 Refer to *fprintf()*.

44025 **NAME**

44026 *sockatmark* — determine whether a socket is at the out-of-band mark

44027 **SYNOPSIS**

```
44028     #include <sys/socket.h>
44029
44030     int sockatmark(int s);
```

44030 **DESCRIPTION**

44031 The *sockatmark*() function shall determine whether the socket specified by the descriptor *s* is at
44032 the out-of-band data mark (see the System Interfaces volume of IEEE Std 1003.1-2001, Section
44033 2.10.12, Socket Out-of-Band Data State). If the protocol for the socket supports out-of-band data
44034 by marking the stream with an out-of-band data mark, the *sockatmark*() function shall return 1
44035 when all data preceding the mark has been read and the out-of-band data mark is the first
44036 element in the receive queue. The *sockatmark*() function shall not remove the mark from the
44037 stream.

44038 **RETURN VALUE**

44039 Upon successful completion, the *sockatmark*() function shall return a value indicating whether
44040 the socket is at an out-of-band data mark. If the protocol has marked the data stream and all data
44041 preceding the mark has been read, the return value shall be 1; if there is no mark, or if data
44042 precedes the mark in the receive queue, the *sockatmark*() function shall return 0. Otherwise, it
44043 shall return a value of -1 and set *errno* to indicate the error.

44044 **ERRORS**

44045 The *sockatmark*() function shall fail if:

44046 [EBADF] The *s* argument is not a valid file descriptor.

44047 [ENOTTY] The *s* argument does not specify a descriptor for a socket.

44048 **EXAMPLES**

44049 None.

44050 **APPLICATION USAGE**

44051 The use of this function between receive operations allows an application to determine which
44052 received data precedes the out-of-band data and which follows the out-of-band data.

44053 There is an inherent race condition in the use of this function. On an empty receive queue, the
44054 current read of the location might well be at the “mark”, but the system has no way of knowing
44055 that the next data segment that will arrive from the network will carry the mark, and
44056 *sockatmark*() will return false, and the next read operation will silently consume the mark.

44057 Hence, this function can only be used reliably when the application already knows that the out-
44058 of-band data has been seen by the system or that it is known that there is data waiting to be read
44059 at the socket (via SIGURG or *select*()). See Section 2.10.11 (on page 61), Section 2.10.12 (on page
44060 62), Section 2.10.14 (on page 63), and *pselect*() for details.

44061 **RATIONALE**

44062 The *sockatmark*() function replaces the historical SIOCATMARK command to *ioctl*() which
44063 implemented the same functionality on many implementations. Using a wrapper function
44064 follows the adopted conventions to avoid specifying commands to the *ioctl*() function, other
44065 than those now included to support XSI STREAMS. The *sockatmark*() function could be
44066 implemented as follows:

```
44067     #include <sys/ioctl.h>
44068
44069     int sockatmark(int s)
44070     {
```

```
44070     int val;
44071     if (ioctl(s,SIOCATMARK,&val)==-1)
44072         return(-1);
44073     return(val);
44074 }
```

44075 The use of [ENOTTY] to indicate an incorrect descriptor type matches the historical behavior of
44076 SIOCATMARK.

44077 FUTURE DIRECTIONS

44078 None.

44079 SEE ALSO

44080 *pselect()*, *recv()*, *recvmsg()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

44081 CHANGE HISTORY

44082 First released in Issue 6. Derived from IEEE Std 1003.1g-2000.

44083 NAME

44084 socket — create an endpoint for communication

44085 SYNOPSIS

```
44086       #include <sys/socket.h>
44087       int socket(int domain, int type, int protocol);
```

44088 DESCRIPTION

44089 The *socket()* function shall create an unbound socket in a communications domain, and return a
44090 file descriptor that can be used in later function calls that operate on sockets.

44091 The *socket()* function takes the following arguments:

44092 *domain* Specifies the communications domain in which a socket is to be created.
44093 *type* Specifies the type of socket to be created.
44094 *protocol* Specifies a particular protocol to be used with the socket. Specifying a *protocol*
44095 of 0 causes *socket()* to use an unspecified default protocol appropriate for the
44096 requested socket type.

44097 The *domain* argument specifies the address family used in the communications domain. The
44098 address families supported by the system are implementation-defined.

44099 Symbolic constants that can be used for the *domain* argument are defined in the <sys/socket.h>
44100 header.

44101 The *type* argument specifies the socket type, which determines the semantics of communication
44102 over the socket. The following socket types are defined; implementations may specify additional
44103 socket types:

44104 SOCK_STREAM Provides sequenced, reliable, bidirectional, connection-mode byte
44105 streams, and may provide a transmission mechanism for out-of-band
44106 data.
44107 SOCK_DGRAM Provides datagrams, which are connectionless-mode, unreliable messages
44108 of fixed maximum length.
44109 SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-mode
44110 transmission paths for records. A record can be sent using one or more
44111 output operations and received using one or more input operations, but a
44112 single operation never transfers part of more than one record. Record
44113 boundaries are visible to the receiver via the MSG_EOR flag.

44114 If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address
44115 family. If the *protocol* argument is zero, the default protocol for this address family and type shall
44116 be used. The protocols supported by the system are implementation-defined.

44117 The process may need to have appropriate privileges to use the *socket()* function or to create
44118 some sockets.

44119 RETURN VALUE

44120 Upon successful completion, *socket()* shall return a non-negative integer, the socket file
44121 descriptor. Otherwise, a value of -1 shall be returned and *errno* set to indicate the error.

44122 ERRORS

44123 The *socket()* function shall fail if:

44124 [EAFNOSUPPORT]
44125 The implementation does not support the specified address family.

- 44126 [EMFILE] No more file descriptors are available for this process.
- 44127 [ENFILE] No more file descriptors are available for the system.
- 44128 [EPROTONOSUPPORT] The protocol is not supported by the address family, or the protocol is not supported by the implementation.
- 44129
44130 [EPROTOTYPE] The socket type is not supported by the protocol.
- 44131 The *socket()* function may fail if:
- 44132 [EACCES] The process does not have appropriate privileges.
- 44133 [ENOBUFS] Insufficient resources were available in the system to perform the operation.
- 44134 [ENOMEM] Insufficient memory was available to fulfill the request.

44136 EXAMPLES

44137 None.

44138 APPLICATION USAGE

44139 The documentation for specific address families specifies which protocols each address family supports. The documentation for specific protocols specifies which socket types each protocol supports.

44142 The application can determine whether an address family is supported by trying to create a socket with *domain* set to the protocol in question.

44144 RATIONALE

44145 None.

44146 FUTURE DIRECTIONS

44147 None.

44148 SEE ALSO

44149 *accept()*, *bind()*, *connect()*, *getsockname()*, *getsockopt()*, *listen()*, *recv()*, *recvfrom()*, *recvmsg()*,
44150 *send()*, *sendmsg()*, *setsockopt()*, *shutdown()*, *socketpair()*, the Base Definitions volume of
44151 IEEE Std 1003.1-2001, <netinet/in.h>, <sys/socket.h>

44152 CHANGE HISTORY

44153 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

44154 NAME

44155 socketpair — create a pair of connected sockets

44156 SYNOPSIS

```
44157       #include <sys/socket.h>
44158       int socketpair(int domain, int type, int protocol,
44159                    int socket_vector[2]);
```

44160 DESCRIPTION

44161 The *socketpair()* function shall create an unbound pair of connected sockets in a specified *domain*,
44162 of a specified *type*, under the protocol optionally specified by the *protocol* argument. The two
44163 sockets shall be identical. The file descriptors used in referencing the created sockets shall be
44164 returned in *socket_vector[0]* and *socket_vector[1]*.

44165 The *socketpair()* function takes the following arguments:

44166 *domain* Specifies the communications domain in which the sockets are to be created.
44167 *type* Specifies the type of sockets to be created.
44168 *protocol* Specifies a particular protocol to be used with the sockets. Specifying a
44169 *protocol* of 0 causes *socketpair()* to use an unspecified default protocol
44170 appropriate for the requested socket type.
44171 *socket_vector* Specifies a 2-integer array to hold the file descriptors of the created socket
44172 pair.

44173 The *type* argument specifies the socket type, which determines the semantics of communications
44174 over the socket. The following socket types are defined; implementations may specify additional
44175 socket types:

44176 SOCK_STREAM Provides sequenced, reliable, bidirectional, connection-mode byte
44177 streams, and may provide a transmission mechanism for out-of-band
44178 data.
44179 SOCK_DGRAM Provides datagrams, which are connectionless-mode, unreliable messages
44180 of fixed maximum length.
44181 SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-mode
44182 transmission paths for records. A record can be sent using one or more
44183 output operations and received using one or more input operations, but a
44184 single operation never transfers part of more than one record. Record
44185 boundaries are visible to the receiver via the MSG_EOR flag.

44186 If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address
44187 family. If the *protocol* argument is zero, the default protocol for this address family and type shall
44188 be used. The protocols supported by the system are implementation-defined.

44189 The process may need to have appropriate privileges to use the *socketpair()* function or to create
44190 some sockets.

44191 RETURN VALUE

44192 Upon successful completion, this function shall return 0; otherwise, -1 shall be returned and
44193 *errno* set to indicate the error.

44194 ERRORS

44195 The *socketpair()* function shall fail if:

44196 [EAFNOSUPPORT] The implementation does not support the specified address family.
44197

44198 [EMFILE] No more file descriptors are available for this process.
44199 [ENFILE] No more file descriptors are available for the system.
44200 [EOPNOTSUPP] The specified protocol does not permit creation of socket pairs.
44201 [EPROTONOSUPPORT]
44202 The protocol is not supported by the address family, or the protocol is not
44203 supported by the implementation.
44204 [EPROTOTYPE] The socket type is not supported by the protocol.
44205 The *socketpair()* function may fail if:
44206 [EACCES] The process does not have appropriate privileges.
44207 [ENOBUFS] Insufficient resources were available in the system to perform the operation.
44208 [ENOMEM] Insufficient memory was available to fulfill the request.

44209 EXAMPLES

44210 None.

44211 APPLICATION USAGE

44212 The documentation for specific address families specifies which protocols each address family
44213 supports. The documentation for specific protocols specifies which socket types each protocol
44214 supports.

44215 The *socketpair()* function is used primarily with UNIX domain sockets and need not be
44216 supported for other domains.

44217 RATIONALE

44218 None.

44219 FUTURE DIRECTIONS

44220 None.

44221 SEE ALSO

44222 *socket()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

44223 CHANGE HISTORY

44224 First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

44225 NAME

44226 `sprintf` — print formatted output

44227 SYNOPSIS

44228 `#include <stdio.h>`

44229 `int sprintf(char *restrict s, const char *restrict format, ...);`

44230 DESCRIPTION

44231 Refer to *fprintf()*.

44232 NAME

44233 sqrt, sqrtf, sqrtl — square root function

44234 SYNOPSIS

```
44235       #include <math.h>
44236       double sqrt(double x);
44237       float sqrtf(float x);
44238       long double sqrtl(long double x);
```

44239 DESCRIPTION

44240 CX 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.

44243 These functions shall compute the square root of their argument x , \sqrt{x} .

44244 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.

44248 RETURN VALUE

44249 Upon successful completion, these functions shall return the square root of x .

44250 MX For finite values of $x < -0$, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

44252 MX If x is NaN, a NaN shall be returned.

44253 If x is ± 0 or $+Inf$, x shall be returned.

44254 If x is $-Inf$, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

44256 ERRORS

44257 These functions shall fail if:

44258 MX Domain Error The finite value of x is < -0 , or x is $-Inf$.

44259 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.

44263 EXAMPLES**44264 Taking the Square Root of 9.0**

```
44265       #include <math.h>
44266       ...
44267       double x = 9.0;
44268       double result;
44269       ...
44270       result = sqrt(x);
```

44271 APPLICATION USAGE

44272 On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling &
44273 MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

44274 RATIONALE

44275 None.

44276 FUTURE DIRECTIONS

44277 None.

44278 SEE ALSO

44279 *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001,
44280 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>, <**stdio.h**>

44281 CHANGE HISTORY

44282 First released in Issue 1. Derived from Issue 1 of the SVID.

44283 Issue 5

44284 The DESCRIPTION is updated to indicate how an application should check for an error. This
44285 text was previously published in the APPLICATION USAGE section.

44286 Issue 6

44287 The *sqrtf()* and *sqrtl()* functions are added for alignment with the ISO/IEC 9899:1999 standard.

44288 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
44289 revised to align with the ISO/IEC 9899:1999 standard.

44290 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are
44291 marked.

44292 NAME

44293 **srand** — pseudo-random number generator

44294 SYNOPSIS

44295 #include <stdlib.h>

44296 void srand(unsigned seed);

44297 DESCRIPTION

44298 Refer to *rand()*.

44299 NAME

44300 **rand48** — seed the uniformly distributed double-precision pseudo-random number generator

44301 SYNOPSIS

44302 XSI

```
#include <stdlib.h>
```

44303

```
void srand48(long seedval);
```

44304

44305 DESCRIPTION

44306 Refer to *drand48()*.

44307 NAME

44308 **srandom** — seed pseudo-random number generator

44309 SYNOPSIS

44310 XSI #include <stdlib.h>

44311 void srandom(unsigned seed);

44312

44313 DESCRIPTION

44314 Refer to *initstate()*.

44315 **NAME**44316 **sscanf** — convert formatted input44317 **SYNOPSIS**

44318 #include <stdio.h>

44319 int sscanf(const char *restrict *s*, const char *restrict *format*, ...);44320 **DESCRIPTION**44321 Refer to *fscanf*().

44322 NAME

44323 stat — get file status

44324 SYNOPSIS

44325 #include <sys/stat.h>
44326 int stat(const char *restrict path, struct stat *restrict buf);

44327 DESCRIPTION

44328 The *stat()* function shall obtain information about the named file and write it to the area pointed to by the *buf* argument. The *path* argument points to a pathname naming a file. Read, write, or execute permission of the named file is not required. An implementation that provides additional or alternate file access control mechanisms may, under implementation-defined conditions, cause *stat()* to fail. In particular, the system may deny the existence of the file specified by *path*.

44334 If the named file is a symbolic link, the *stat()* function shall continue pathname resolution using the contents of the symbolic link, and shall return information pertaining to the resulting file if the file exists.

44337 The *buf* argument is a pointer to a **stat** structure, as defined in the **<sys/stat.h>** header, into which information is placed concerning the file.

44339 The *stat()* function shall update any time-related fields (as described in the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.7, File Times Update), before writing into the **stat** structure.

44342 Unless otherwise specified, the structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime* shall have meaningful values for all file types defined in this volume of IEEE Std 1003.1-2001. The value of the member *st_nlink* shall be set to the number of links to the file.

44346 RETURN VALUE

44347 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

44349 ERRORS

44350 The *stat()* function shall fail if:

- 44351 [EACCES] Search permission is denied for a component of the path prefix.
- 44352 [EIO] An error occurred while reading from the file system.
- 44353 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.
- 44355 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
- 44358 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.
- 44359 [ENOTDIR] A component of the path prefix is not a directory.
- 44360 [EOVERFLOW] The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

44363 The **stat()** function may fail if:

44364 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
44365 resolution of the *path* argument.

44366 [ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the *path* argument,
44367 the length of the substituted pathname string exceeded {PATH_MAX}.

44369 [EOVERFLOW] A value to be stored would overflow one of the members of the **stat** structure.

44370 EXAMPLES

44371 Obtaining File Status Information

44372 The following example shows how to obtain file status information for a file named
44373 **/home/cnd/mod1**. The structure variable *buffer* is defined for the **stat** structure.

```
44374        #include <sys/types.h>
44375        #include <sys/stat.h>
44376        #include <fcntl.h>

44377        struct stat buffer;
44378        int            status;
44379        ...
44380        status = stat( "/home/cnd/mod1" , &buffer );
```

44381 Getting Directory Information

44382 The following example fragment gets status information for each entry in a directory. The call to
44383 the **stat()** function stores file information in the **stat** structure pointed to by *statbuf*. The lines
44384 that follow the **stat()** call format the fields in the **stat** structure for presentation to the user of the
44385 program.

```
44386        #include <sys/types.h>
44387        #include <sys/stat.h>
44388        #include <dirent.h>
44389        #include <pwd.h>
44390        #include <grp.h>
44391        #include <time.h>
44392        #include <locale.h>
44393        #include <langinfo.h>
44394        #include <stdio.h>
44395        #include <stdint.h>

44396        struct dirent *dp;
44397        struct stat    statbuf;
44398        struct passwd *pwd;
44399        struct group   grp;
44400        struct tm     tm;
44401        char          datestring[ 256 ];
44402        ...
44403        /* Loop through directory entries. */
44404        while ((dp = readdir(dir)) != NULL) {

44405                /* Get entry's information. */
44406                if (stat(dp->d_name, &statbuf) == -1)
```

```

44407         continue;

44408     /* Print out type, permissions, and number of links. */
44409     printf("%10.10s", sperm (statbuf.st_mode));
44410     printf("%4d", statbuf.st_nlink);

44411     /* Print out owner's name if it is found using getpwuid(). */
44412     if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
44413         printf(" %-8.8s", pwd->pw_name);
44414     else
44415         printf(" %-8d", statbuf.st_uid);

44416     /* Print out group name if it is found using getgrgid(). */
44417     if ((grp = getgrgid(statbuf.st_gid)) != NULL)
44418         printf(" %-8.8s", grp->gr_name);
44419     else
44420         printf(" %-8d", statbuf.st_gid);

44421     /* Print size of file. */
44422     printf(" %9jd", (intmax_t)statbuf.st_size);

44423     tm = localtime(&statbuf.st_mtime);

44424     /* Get localized date string. */
44425     strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);
44426     printf(" %s %s\n", datestring, dp->d_name);
44427 }

```

44428 APPLICATION USAGE

44429 None.

44430 RATIONALE

44431 The intent of the paragraph describing “additional or alternate file access control mechanisms”
 44432 is to allow a secure implementation where a process with a label that does not dominate the
 44433 file’s label cannot perform a *stat()* function. This is not related to read permission; a process with
 44434 a label that dominates the file’s label does not need read permission. An implementation that
 44435 supports write-up operations could fail *fstat()* function calls even though it has a valid file
 44436 descriptor open for writing.

44437 FUTURE DIRECTIONS

44438 None.

44439 SEE ALSO

44440 *fstat()*, *lstat()*, *readlink()*, *symlink()*, the Base Definitions volume of IEEE Std 1003.1-2001,
 44441 <sys/stat.h>, <sys/types.h>

44442 CHANGE HISTORY

44443 First released in Issue 1. Derived from Issue 1 of the SVID.

44444 Issue 5

44445 Large File Summit extensions are added.

44446 Issue 6

44447 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

44448 The following new requirements on POSIX implementations derive from alignment with the
 44449 Single UNIX Specification:

44450 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
44451 required for conforming implementations of previous POSIX specifications, it was not
44452 required for UNIX applications.

44453 • The [EIO] mandatory error condition is added.

44454 • The [ELOOP] mandatory error condition is added.

44455 • The [EOVERFLOW] mandatory error condition is added. This change is to support large
44456 files.

44457 • The [ENAMETOOLONG] and the second [EOVERFLOW] optional error conditions are
44458 added.

44459 The following changes were made to align with the IEEE P1003.1a draft standard:

44460 • Details are added regarding the treatment of symbolic links.

44461 • The [ELOOP] optional error condition is added.

44462 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

44463 The **restrict** keyword is added to the *stat()* prototype for alignment with the ISO/IEC 9899: 1999
44464 standard.

44465 NAME

44466 statvfs — get file system information

44467 SYNOPSIS

44468 XSI #include <sys/statvfs.h>

44469 int statvfs(const char *restrict *path*, struct statvfs *restrict *buf*);

44470

44471 DESCRIPTION

44472 Refer to *fstatvfs()*.

44473 NAME

44474 `stderr, stdin, stdout` — standard I/O streams

44475 SYNOPSIS

```
44476        #include <stdio.h>  
44477  
44478        extern FILE *stderr, *stdin, *stdout;
```

44478 DESCRIPTION

44479 CX 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.

44482 A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE**. The *fopen()* function shall create certain descriptive data for a stream and return a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **<stdio.h>** header and associated with the standard open files.

44486 At program start-up, three streams shall be 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.

44491 CX The following symbolic values in **<unistd.h>** define the file descriptors that shall be associated with the C-language *stdin*, *stdout*, and *stderr* when the application is started:

44493 **STDIN_FILENO** Standard input value, *stdin*. Its value is 0.

44494 **STDOUT_FILENO** Standard output value, *stdout*. Its value is 1.

44495 **STDERR_FILENO** Standard error value, *stderr*. Its value is 2.

44496 The *stderr* stream is expected to be open for reading and writing.

44497 RETURN VALUE

44498 None.

44499 ERRORS

44500 No errors are defined.

44501 EXAMPLES

44502 None.

44503 APPLICATION USAGE

44504 None.

44505 RATIONALE

44506 None.

44507 FUTURE DIRECTIONS

44508 None.

44509 SEE ALSO

44510 *fclose()*, *feof()*, *fferror()*, *fileno()*, *fopen()*, *fread()*, *fseek()*, *getc()*, *gets()*, *popen()*, *printf()*, *putc()*,
44511 *puts()*, *read()*, *scanf()*, *setbuf()*, *setvbuf()*, *tmpfile()*, *ungetc()*, *vprintf()*, the Base Definitions
44512 volume of IEEE Std 1003.1-2001, **<stdio.h>**, **<unistd.h>**

44513 CHANGE HISTORY

44514 First released in Issue 1.

44515 Issue 6

44516 Extensions beyond the ISO C standard are marked.

44517 A note that *stderr* is expected to be open for reading and writing is added to the DESCRIPTION.

44518 NAME

44519 **strcasecmp, strncasecmp** — case-insensitive string comparisons

44520 SYNOPSIS

44521 XSI `#include <strings.h>`

```
44522       int strcasecmp(const char *s1, const char *s2);  
44523       int strncasecmp(const char *s1, const char *s2, size_t n);  
44524
```

44525 DESCRIPTION

44526 The *strcasecmp()* function shall compare, while ignoring differences in case, the string pointed to by *s1* to the string pointed to by *s2*. The *strncasecmp()* function shall compare, while ignoring differences in case, not more than *n* bytes from the string pointed to by *s1* to the string pointed to by *s2*.

44530 In the POSIX locale, *strcasecmp()* and *strncasecmp()* shall behave as if the strings had been converted to lowercase and then a byte comparison performed. The results are unspecified in other locales.

44533 RETURN VALUE

44534 Upon completion, *strcasecmp()* shall return an integer greater than, equal to, or less than 0, if the string pointed to by *s1* is, ignoring case, greater than, equal to, or less than the string pointed to by *s2*, respectively.

44537 Upon successful completion, *strncasecmp()* shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by *s1* is, ignoring case, greater than, equal to, or less than the possibly null-terminated array pointed to by *s2*, respectively.

44540 ERRORS

44541 No errors are defined.

44542 EXAMPLES

44543 None.

44544 APPLICATION USAGE

44545 None.

44546 RATIONALE

44547 None.

44548 FUTURE DIRECTIONS

44549 None.

44550 SEE ALSO

44551 The Base Definitions volume of IEEE Std 1003.1-2001, `<strings.h>`

44552 CHANGE HISTORY

44553 First released in Issue 4, Version 2.

44554 Issue 5

44555 Moved from X/OPEN UNIX extension to BASE.

44556 NAME

44557 `strcat` — concatenate two strings

44558 SYNOPSIS

```
44559        #include <string.h>
44560        char *strcat(char *restrict s1, const char *restrict s2);
```

44561 DESCRIPTION

44562 CX 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.

44565 The `strcat()` function shall append a copy of the string pointed to by *s2* (including the terminating null byte) to the end of the string pointed to by *s1*. The initial byte of *s2* overwrites the null byte at the end of *s1*. If copying takes place between objects that overlap, the behavior is undefined.

44569 RETURN VALUE

44570 The `strcat()` function shall return *s1*; no return value is reserved to indicate an error.

44571 ERRORS

44572 No errors are defined.

44573 EXAMPLES

44574 None.

44575 APPLICATION USAGE

44576 This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3 applications. Reliable error detection by this function was never guaranteed.

44578 RATIONALE

44579 None.

44580 FUTURE DIRECTIONS

44581 None.

44582 SEE ALSO

44583 `strncat()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<string.h>`

44584 CHANGE HISTORY

44585 First released in Issue 1. Derived from Issue 1 of the SVID.

44586 Issue 6

44587 The `strcat()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

44588 NAME

44589 `strchr` — string scanning operation

44590 SYNOPSIS

```
44591        #include <string.h>
44592        char *strchr(const char *s, int c);
```

44593 DESCRIPTION

44594 CX 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.

44597 The `strchr()` function shall locate the first occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating null byte is considered to be part of the string.

44599 RETURN VALUE

44600 Upon completion, `strchr()` shall return a pointer to the byte, or a null pointer if the byte was not found.

44602 ERRORS

44603 No errors are defined.

44604 EXAMPLES

44605 None.

44606 APPLICATION USAGE

44607 None.

44608 RATIONALE

44609 None.

44610 FUTURE DIRECTIONS

44611 None.

44612 SEE ALSO

44613 `strrchr()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<string.h>`

44614 CHANGE HISTORY

44615 First released in Issue 1. Derived from Issue 1 of the SVID.

44616 Issue 6

44617 Extensions beyond the ISO C standard are marked.

44618 NAME

44619 strcmp — compare two strings

44620 SYNOPSIS

```
44621       #include <string.h>
44622       int strcmp(const char *s1, const char *s2);
```

44623 DESCRIPTION

44624 CX 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.

44627 The *strcmp()* function shall compare the string pointed to by *s1* to the string pointed to by *s2*.

44628 The sign of a non-zero return value shall be determined by the sign of the difference between the
44629 values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings
44630 being compared.

44631 RETURN VALUE

44632 Upon completion, *strcmp()* shall return an integer greater than, equal to, or less than 0, if the
44633 string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*,
44634 respectively.

44635 ERRORS

44636 No errors are defined.

44637 EXAMPLES

44638 Checking a Password Entry

44639 The following example compares the information read from standard input to the value of the
44640 name of the user entry. If the *strcmp()* function returns 0 (indicating a match), a further check
44641 will be made to see if the user entered the proper old password. The *crypt()* function shall
44642 encrypt the old password entered by the user, using the value of the encrypted password in the
44643 **passwd** structure as the salt. If this value matches the value of the encrypted **passwd** in the
44644 structure, the entered password *oldpasswd* is the correct user's password. Finally, the program
44645 encrypts the new password so that it can store the information in the **passwd** structure.

```
44646       #include <string.h>
44647       #include <unistd.h>
44648       #include <stdio.h>
44649       ...
44650       int valid_change;
44651       struct passwd *p;
44652       char user[100];
44653       char oldpasswd[100];
44654       char newpasswd[100];
44655       char savepasswd[100];
44656       ...
44657       if (strcmp(p->pw_name, user) == 0) {
44658           if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
44659               strcpy(savepasswd, crypt(newpasswd, user));
44660               p->pw_passwd = savepasswd;
44661               valid_change = 1;
44662           }
44663       else {
```

```
44664         fprintf(stderr, "Old password is not valid\n");
44665     }
44666 }
44667 . . .
```

44668 APPLICATION USAGE

44669 None.

44670 RATIONALE

44671 None.

44672 FUTURE DIRECTIONS

44673 None.

44674 SEE ALSO

44675 *strncmp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>

44676 CHANGE HISTORY

44677 First released in Issue 1. Derived from Issue 1 of the SVID.

44678 Issue 6

44679 Extensions beyond the ISO C standard are marked.

44680 NAME

44681 *strcoll* — string comparison using collating information

44682 SYNOPSIS

```
44683        #include <string.h>
44684        int strcoll(const char *s1, const char *s2);
```

44685 DESCRIPTION

44686 CX 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.

44689 The *strcoll()* function shall compare the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the *LC_COLLATE* category of the current locale.

44691 CX The *strcoll()* function shall not change the setting of *errno* if successful.

44692 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 *strcoll()*, then check *errno*.

44694 RETURN VALUE

44695 Upon successful completion, *strcoll()* shall return an integer greater than, equal to, or less than 0, according to whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2* when both are interpreted as appropriate to the current locale. On error, *strcoll()* may set *errno*, but no return value is reserved to indicate an error.

44699 ERRORS

44700 The *strcoll()* function may fail if:

44701 CX [EINVAL] The *s1* or *s2* arguments contain characters outside the domain of the collating sequence.

44703 EXAMPLES**44704 Comparing Nodes**

44705 The following example uses an application-defined function, *node_compare()*, to compare two nodes based on an alphabetical ordering of the *string* field.

```
44707        #include <string.h>
44708        ...
44709        struct node { /* These are stored in the table. */
44710            char *string;
44711            int length;
44712        };
44713        ...
44714        int node_compare(const void *node1, const void *node2)
44715        {
44716            return strcoll(((const struct node *)node1)->string,
44717                            ((const struct node *)node2)->string);
44718        }
44719        ...
```

44720 APPLICATION USAGE

44721 The *strxfrm()* and *strcmp()* functions should be used for sorting large lists.

44722 RATIONALE

44723 None.

44724 FUTURE DIRECTIONS

44725 None.

44726 SEE ALSO

44727 *strcmp()*, *strxfrm()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

44728 CHANGE HISTORY

44729 First released in Issue 3.

44730 Issue 5

44731 The DESCRIPTION is updated to indicate that *errno* does not change if the function is successful.
44732

44733 Issue 6

44734 Extensions beyond the ISO C standard are marked.

44735 The following new requirements on POSIX implementations derive from alignment with the
44736 Single UNIX Specification:

- The [EINVAL] optional error condition is added.

44738 An example is added.

44739 NAME

44740 strcpy — copy a string

44741 SYNOPSIS

44742 #include <string.h>

44743 char *strcpy(char *restrict s1, const char *restrict s2);

44744 DESCRIPTION

44745 CX 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.

44748 The *strcpy()* function shall copy the string pointed to by *s2* (including the terminating null byte) into the array pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

44751 RETURN VALUE

44752 The *strcpy()* function shall return *s1*; no return value is reserved to indicate an error.

44753 ERRORS

44754 No errors are defined.

44755 EXAMPLES

44756 **Initializing a String**

44757 The following example copies the string "-----" into the *permstring* variable.

```
44758       #include <string.h>
44759       ...
44760       static char permstring[11];
44761       ...
44762       strcpy(permstring, "-----");
44763       ...
```

44764 **Storing a Key and Data**

44765 The following example allocates space for a key using *malloc()* then uses *strcpy()* to place the key there. Then it allocates space for data using *malloc()*, and uses *strcpy()* to place data there. (The user-defined function *dbfree()* frees memory previously allocated to an array of type **struct element** *.)

```
44769       #include <string.h>
44770       #include <stdlib.h>
44771       #include <stdio.h>
44772       ...
44773       /* Structure used to read data and store it. */
44774       struct element {
44775           char *key;
44776           char *data;
44777       };
44778       struct element *tbl, *curtbl;
44779       char *key, *data;
44780       int count;
44781       ...
44782       void dbfree(struct element *, int);
```

```
44783     ...
44784     if ((curtbl->key = malloc(strlen(key) + 1)) == NULL) {
44785         perror("malloc");
44786         dbfree(tbl, count);
44787         return NULL;
44788     }
44789     strcpy(curtbl->key, key);
44790
44791     if ((curtbl->data = malloc(strlen(data) + 1)) == NULL) {
44792         perror("malloc");
44793         free(curtbl->key);
44794         dbfree(tbl, count);
44795         return NULL;
44796     }
44797     strcpy(curtbl->data, data);
44798     ...
44799 
```

44793 APPLICATION USAGE

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3 applications. Reliable error detection by this function was never guaranteed.

44798 RATIONALE

44799 None.

44800 FUTURE DIRECTIONS

44801 None.

44802 SEE ALSO

44803 *strncpy()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>

44804 CHANGE HISTORY

44805 First released in Issue 1. Derived from Issue 1 of the SVID.

44806 Issue 6

44807 The *strcpy()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

44808 NAME

44809 strcspn — get the length of a complementary substring

44810 SYNOPSIS

```
44811 #include <string.h>
44812 size_t strcspn(const char *s1, const char *s2);
```

44813 DESCRIPTION

44814 CX 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.

44817 The *strcspn()* function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by *s1* which consists entirely of bytes *not* from the string pointed to by *s2*.

44819 RETURN VALUE

44820 The *strcspn()* function shall return the length of the computed segment of the string pointed to by *s1*; no return value is reserved to indicate an error.

44822 ERRORS

44823 No errors are defined.

44824 EXAMPLES

44825 None.

44826 APPLICATION USAGE

44827 None.

44828 RATIONALE

44829 None.

44830 FUTURE DIRECTIONS

44831 None.

44832 SEE ALSO

44833 *strspn()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

44834 CHANGE HISTORY

44835 First released in Issue 1. Derived from Issue 1 of the SVID.

44836 Issue 5

44837 The RETURN VALUE section is updated to indicate that *strcspn()* returns the length of *s1*, and not *s1* itself as was previously stated.

44839 Issue 6

44840 The Open Group Corrigendum U030/1 is applied. The text of the RETURN VALUE section is updated to indicate that the computed segment length is returned, not the *s1* length.

44842 NAME

44843 *strup* — duplicate a string

44844 SYNOPSIS

44845 XSI #include <string.h>

44846 char *strup(const char *s1);

44847

44848 DESCRIPTION

44849 The *strup()* function shall return a pointer to a new string, which is a duplicate of the string
44850 pointed to by *s1*. The returned pointer can be passed to *free()*. A null pointer is returned if the
44851 new string cannot be created.

44852 RETURN VALUE

44853 The *strup()* function shall return a pointer to a new string on success. Otherwise, it shall return
44854 a null pointer and set *errno* to indicate the error.

44855 ERRORS

44856 The *strup()* function may fail if:

44857 [ENOMEM] Storage space available is insufficient.

44858 EXAMPLES

44859 None.

44860 APPLICATION USAGE

44861 None.

44862 RATIONALE

44863 None.

44864 FUTURE DIRECTIONS

44865 None.

44866 SEE ALSO

44867 *free()*, *malloc()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

44868 CHANGE HISTORY

44869 First released in Issue 4, Version 2.

44870 Issue 5

44871 Moved from X/OPEN UNIX extension to BASE.

44872 NAME

44873 `strerror, strerror_r — get error message string`

44874 SYNOPSIS

```
44875        #include <string.h>
44876        char *strerror(int errnum);
44877 TSF        int strerror_r(int errnum, char *strerrbuf, size_t buflen);
```

44879 DESCRIPTION

44880 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.

44883 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.

44886 The string pointed to shall not be modified by the application, but may be overwritten by a subsequent call to `strerror()` or `perror()`.

44888 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.

44890 The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-2001 calls `strerror()`.

44892 CX The `strerror()` function shall not change the setting of `errno` if successful.

44893 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`.

44895 The `strerror()` function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

44897 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`.

44900 RETURN VALUE

44901 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.

44903 TSF Upon successful completion, `strerror_r()` shall return 0. Otherwise, an error number shall be returned to indicate the error.

44905 ERRORS

44906 These functions may fail if:

44907 [EINVAL] The value of `errnum` is not a valid error number.

44908 The `strerror_r()` function may fail if:

44909 TSF [ERANGE] Insufficient storage was supplied via `strerrbuf` and `buflen` to contain the generated message string.

44911 EXAMPLES

44912 None.

44913 APPLICATION USAGE

44914 None.

44915 RATIONALE

44916 None.

44917 FUTURE DIRECTIONS

44918 None.

44919 SEE ALSO

44920 *perror()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

44921 CHANGE HISTORY

44922 First released in Issue 3.

44923 Issue 5

44924 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

44925 A note indicating that this function need not be reentrant is added to the DESCRIPTION.

44926 Issue 6

44927 Extensions beyond the ISO C standard are marked.

44928 The following new requirements on POSIX implementations derive from alignment with the
44929 Single UNIX Specification:

44930 • In the RETURN VALUE section, the fact that *errno* may be set is added.

44931 • The [EINVAL] optional error condition is added.

44932 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

44933 The *strerror_r()* function is added in response to IEEE PASC Interpretation 1003.1c #39.

44934 The *strerror_r()* function is marked as part of the Thread-Safe Functions option.

44935 NAME

44936 strfmon — convert monetary value to a string

44937 SYNOPSIS

44938 XSI #include <monetary.h>

44939 ssize_t strfmon(char *restrict s, size_t maxsize,
44940 const char *restrict format, ...);

44941

44942 DESCRIPTION

44943 The *strfmon()* function shall place characters into the array pointed to by *s* as controlled by the
44944 string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

44945 The format is a character string, beginning and ending in its initial state, if any, that contains two
44946 types of objects: *plain characters*, which are simply copied to the output stream, and *conversion
44947 specifications*, each of which shall result in the fetching of zero or more arguments which are
44948 converted and formatted. The results are undefined if there are insufficient arguments for the
44949 format. If the format is exhausted while arguments remain, the excess arguments are simply
44950 ignored.

44951 The application shall ensure that a conversion specification consists of the following sequence:

- 44952 • A '%' character
- 44953 • Optional flags
- 44954 • Optional field width
- 44955 • Optional left precision
- 44956 • Optional right precision
- 44957 • A required conversion specifier character that determines the conversion to be performed

44958 Flags

44959 One or more of the following optional flags can be specified to control the conversion:

- 44960 =*f* An '=' followed by a single character *f* which is used as the numeric fill character. In
44961 order to work with precision or width counts, the fill character shall be a single byte
44962 character; if not, the behavior is undefined. The default numeric fill character is the
44963 <space>. This flag does not affect field width filling which always uses the <space>. This
44964 flag is ignored unless a left precision (see below) is specified.
- 44965 ^ Do not format the currency amount with grouping characters. The default is to insert
44966 the grouping characters if defined for the current locale.
- 44967 + or (Specify the style of representing positive and negative currency amounts. Only one of
44968 '+' or '(' may be specified. If '+' is specified, the locale's equivalent of '+' and '-'
44969 are used (for example, in the U.S., the empty string if positive and '-' if negative). If
44970 '(' is specified, negative amounts are enclosed within parentheses. If neither flag is
44971 specified, the '+' style is used.
- 44972 ! Suppress the currency symbol from the output conversion.
- 44973 - Specify the alignment. If this flag is present the result of the conversion is left-justified
44974 (padded to the right) rather than right-justified. This flag shall be ignored unless a field
44975 width (see below) is specified.

44976

Field Width

44977

- w A decimal digit string *w* specifying a minimum field width in bytes in which the result of the conversion is right-justified (or left-justified if the flag '-' is specified). The default is 0.

44980

Left Precision

44981

- #*n* A '#' followed by a decimal digit string *n* specifying a maximum number of digits expected to be formatted to the left of the radix character. This option can be used to keep the formatted output from multiple calls to the *strfmon()* function aligned in the same columns. It can also be used to fill unused positions with a special character as in "\$***123.45". This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more than *n* digit positions are required, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character (see the =*f*flag above).

44989

44990

44991

If grouping has not been suppressed with the '^' flag, and it is defined for the current locale, grouping separators are inserted before the fill characters (if any) are added. Grouping separators are not applied to fill characters even if the fill character is a digit.

44992

44993

44994

To ensure alignment, any characters appearing before or after the number in the formatted output such as currency or sign symbols are padded as necessary with <space>s to make their positive and negative formats an equal length.

44995

Right Precision

44996

- .*p* A period followed by a decimal digit string *p* specifying the number of digits after the radix character. If the value of the right precision *p* is 0, no radix character appears. If a right precision is not included, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

45001

ConversionSpecifier Characters

45002

The conversion specifier characters and their meanings are:

45003

- i The **double** argument is formatted according to the locale's international currency format (for example, in the U.S.: USD 1,234.56). If the argument is ±Inf or NaN, the result of the conversion is unspecified.

45006

- n The **double** argument is formatted according to the locale's national currency format (for example, in the U.S.: \$1,234.56). If the argument is ±Inf or NaN, the result of the conversion is unspecified.

45009

- % Convert to a '%'; no argument is converted. The entire conversion specification shall be %%.

45011

Locale Information

45012

The *LC_MONETARY* category of the program's locale affects the behavior of this function including the monetary radix character (which may be different from the numeric radix character affected by the *LC_NUMERIC* category), the grouping separator, the currency symbols, and formats. The international currency symbol should be conformant with the ISO 4217: 2001 standard.

45017

If the value of *maxsize* is greater than {SSIZE_MAX}, the result is implementation-defined.

1

45018 RETURN VALUE

45019 If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, *strfmon()* shall return the number of bytes placed into the array pointed to by *s*, not
 45020 including the terminating null byte. Otherwise, -1 shall be returned, the contents of the array are
 45021 unspecified, and *errno* shall be set to indicate the error.

45023 ERRORS

45024 The *strfmon()* function shall fail if:

45025 [E2BIG] Conversion stopped due to lack of space in the buffer.

45026 EXAMPLES

45027 Given a locale for the U.S. and the values 123.45, -123.45, and 3456.781, the following output
 45028 might be produced. Square brackets ("[]") are used in this example to delimit the output.

45029 %n	[\$123.45]	Default formatting
45030	[-\$123.45]	
45031	[\$3,456.78]	
45032 %11n	[\$123.45]	Right align within an 11-character field
45033	[-\$123.45]	
45034	[\$3,456.78]	
45035 %#5n	[\$ 123.45]	Aligned columns for values up to 99 999
45036	[-\$ 123.45]	
45037	[\$ 3,456.78]	
45038 %-*#5n	[\$****123.45]	Specify a fill character
45039	[-\$****123.45]	
45040	[\$*3,456.78]	
45041 %=0#5n	[\$000123.45]	Fill characters do not use grouping even if the fill character is a digit
45042	[-\$000123.45]	
45043	[\$03,456.78]	
45044 %^#5n	[\$ 123.45]	Disable the grouping separator
45045	[-\$ 123.45]	
45046	[\$ 3456.78]	
45047 %^#5.0n	[\$ 123]	Round off to whole units
45048	[-\$ 123]	
45049	[\$ 3457]	
45050 %^#5.4n	[\$ 123.4500]	Increase the precision
45051	[-\$ 123.4500]	
45052	[\$ 3456.7810]	
45053 %(#5n	[\$(123.45]	Use an alternative pos/neg style
45054	[(\$ 123.45)]	
45055	[\$ 3,456.78]	
45056 %!(#5n	[123.45]	Disable the currency symbol
45057	[(123.45)]	
45058	[3,456.78]	
45059 %-14#5.4n	[\$ 123.4500]	Left-justify the output
45060	[-\$ 123.4500]	
45061	[\$ 3,456.7810]	

45062 %14#5.4n [\$ 123.4500] Corresponding right-justified output
45063 [-\$ 123.4500]
45064 [\$ 3,456.7810]

45065 See also the EXAMPLES section in *fprintf()*.

45066 APPLICATION USAGE

45067 None.

45068 RATIONALE

45069 None.

45070 FUTURE DIRECTIONS

45071 Lowercase conversion characters are reserved for future standards use and uppercase for
45072 implementation-defined use.

45073 SEE ALSO

45074 *fprintf()*, *localeconv()*, the Base Definitions volume of IEEE Std 1003.1-2001, <monetary.h>

45075 CHANGE HISTORY

45076 First released in Issue 4.

45077 Issue 5

45078 Moved from ENHANCED I18N to BASE.

45079 The [ENOSYS] error is removed.

45080 A sentence is added to the DESCRIPTION warning about values of *maxsize* that are greater than
45081 {SSIZE_MAX}.

45082 Issue 6

45083 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

45084 The **restrict** keyword is added to the *strfmon()* prototype for alignment with the
45085 ISO/IEC 9899:1999 standard.

45086 The EXAMPLES section is reworked, clarifying the output format.

45087 NAME

45088 strftime — convert date and time to a string

45089 SYNOPSIS

```
45090     #include <time.h>
45091
45092     size_t strftime(char *restrict s, size_t maxsize,
45093                     const char *restrict format, const struct tm *restrict timeptr);
```

45093 DESCRIPTION

45094 CX 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.

45097 The *strftime()* function shall place bytes into the array pointed to by *s* as controlled by the string pointed to by *format*. The format is a character string, beginning and ending in its initial shift state, if any. The *format* string consists of zero or more conversion specifications and ordinary characters. A conversion specification consists of a '%' character, possibly followed by an E or O modifier, and a terminating conversion specifier character that determines the conversion specification's behavior. All ordinary characters (including the terminating null byte) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than *maxsize* bytes are placed into the array. Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined using the *LC_TIME* category of the current locale and by the values of zero or more members of the broken-down time structure pointed to by *timeptr*, as specified in brackets in the description. If any of the specified values are outside the normal range, the characters stored are unspecified.

45110 CX Local timezone information is used as though *strftime()* called *tzset()*.

45111 The following conversion specifications are supported:

45112 %a	Replaced by the locale's abbreviated weekday name. [<i>tm_wday</i>]
45113 %A	Replaced by the locale's full weekday name. [<i>tm_wday</i>]
45114 %b	Replaced by the locale's abbreviated month name. [<i>tm_mon</i>]
45115 %B	Replaced by the locale's full month name. [<i>tm_mon</i>]
45116 %c	Replaced by the locale's appropriate date and time representation. (See the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>.)
45118 %C	Replaced by the year divided by 100 and truncated to an integer, as a decimal number [00,99]. [<i>tm_year</i>]
45120 %d	Replaced by the day of the month as a decimal number [01,31]. [<i>tm_mday</i>]
45121 %D	Equivalent to %m/%d/%y. [<i>tm_mon</i> , <i>tm_mday</i> , <i>tm_year</i>]
45122 %e	Replaced by the day of the month as a decimal number [1,31]; a single digit is preceded by a space. [<i>tm_mday</i>]
45124 %F	Equivalent to %Y-%m-%d (the ISO 8601:2000 standard date format). [<i>tm_year</i> , <i>tm_mon</i> , <i>tm_mday</i>]
45126 %g	Replaced by the last 2 digits of the week-based year (see below) as a decimal number [00,99]. [<i>tm_year</i> , <i>tm_wday</i> , <i>tm_yday</i>]
45128 %G	Replaced by the week-based year (see below) as a decimal number (for example, 1977). [<i>tm_year</i> , <i>tm_wday</i> , <i>tm_yday</i>]

45130	%h	Equivalent to %b. [tm_mon]
45131	%H	Replaced by the hour (24-hour clock) as a decimal number [00,23]. [tm_hour]
45132	%I	Replaced by the hour (12-hour clock) as a decimal number [01,12]. [tm_hour]
45133	%j	Replaced by the day of the year as a decimal number [001,366]. [tm_yday]
45134	%m	Replaced by the month as a decimal number [01,12]. [tm_mon]
45135	%M	Replaced by the minute as a decimal number [00,59]. [tm_min]
45136	%n	Replaced by a <newline>.
45137	%p	Replaced by the locale's equivalent of either a.m. or p.m. [tm_hour]
45138 CX	%r	Replaced by the time in a.m. and p.m. notation; in the POSIX locale this shall be equivalent to %I:%M:%S %p. [tm_hour, tm_min, tm_sec]
45139		
45140	%R	Replaced by the time in 24-hour notation (%H:%M). [tm_hour, tm_min]
45141	%S	Replaced by the second as a decimal number [00,60]. [tm_sec]
45142	%t	Replaced by a <tab>.
45143	%T	Replaced by the time (%H:%M:%S). [tm_hour, tm_min, tm_sec]
45144	%u	Replaced by the weekday as a decimal number [1,7], with 1 representing Monday. [tm_wday]
45145		
45146	%U	Replaced by the week number of the year as a decimal number [00,53]. The first Sunday of January is the first day of week 1; days in the new year before this are in week 0. [tm_year, tm_wday, tm_yday]
45147		
45148		
45149	%V	Replaced by the week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1. Both January 4th and the first Thursday of January are always in week 1. [tm_year, tm_wday, tm_yday]
45150		
45151		
45152		
45153		
45154	%w	Replaced by the weekday as a decimal number [0,6], with 0 representing Sunday. [tm_wday]
45155		
45156	%W	Replaced by the week number of the year as a decimal number [00,53]. The first Monday of January is the first day of week 1; days in the new year before this are in week 0. [tm_year, tm_wday, tm_yday]
45157		
45158		
45159	%x	Replaced by the locale's appropriate date representation. (See the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>.)
45160		
45161	%X	Replaced by the locale's appropriate time representation. (See the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>.)
45162		
45163	%y	Replaced by the last two digits of the year as a decimal number [00,99]. [tm_year]
45164	%Y	Replaced by the year as a decimal number (for example, 1997). [tm_year]
45165	%z	Replaced by the offset from UTC in the ISO 8601:2000 standard format (+hhmm or -hhmm), or by no characters if no timezone is determinable. For example, "-0430" means 4 hours 30 minutes behind UTC (west of Greenwich). If tm_isdst is zero, the standard time offset is used. If tm_isdst is greater than zero, the daylight savings time offset is used. If tm_isdst is negative, no characters are returned. [tm_isdst]
45166		
45167 CX		
45168		
45169		

45170	%Z	Replaced by the timezone name or abbreviation, or by no bytes if no timezone information exists. [tm_isdst]	
45171			
45172	%%	Replaced by %.	
45173		If a conversion specification does not correspond to any of the above, the behavior is undefined.	
45174	CX	If a struct tm broken-down time structure is created by <i>localtime()</i> or <i>localtime_r()</i> , or modified by <i>mktimel()</i> , and the value of TZ is subsequently modified, the results of the %Z and %z <i>strftime()</i> conversion specifiers are undefined, when <i>strftime()</i> is called with such a broken-down time structure.	1
45175			1
45176			1
45177			1
45178		If a struct tm broken-down time structure is created or modified by <i>gmtimel()</i> or <i>gmtimel_r()</i> , it is unspecified whether the result of the %Z and %z conversion specifiers shall refer to UTC or the current local timezone, when <i>strftime()</i> is called with such a broken-down time structure.	1
45179			1
45180			1

45181 Modified Conversion Specifiers

45182 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.

45187	%Ec	Replaced by the locale's alternative appropriate date and time representation.	
45188	%Ec	Replaced by the name of the base year (period) in the locale's alternative representation.	
45189			
45190	%Ex	Replaced by the locale's alternative date representation.	
45191	%Ex	Replaced by the locale's alternative time representation.	
45192	%Ey	Replaced by the offset from %Ec (year only) in the locale's alternative representation.	
45193	%Ey	Replaced by the full alternative year representation.	
45194	%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.	
45195			
45196			
45197	%Oe	Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.	
45198			
45199	%Oh	Replaced by the hour (24-hour clock) using the locale's alternative numeric symbols.	
45200	%OI	Replaced by the hour (12-hour clock) using the locale's alternative numeric symbols.	
45201	%Om	Replaced by the month using the locale's alternative numeric symbols.	
45202	%OM	Replaced by the minutes using the locale's alternative numeric symbols.	
45203	%OS	Replaced by the seconds using the locale's alternative numeric symbols.	
45204	%Ou	Replaced by the weekday as a number in the locale's alternative representation (Monday=1).	
45205			
45206	%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.	
45207			
45208	%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.	
45209			

45210 %Ow Replaced by the number of the weekday (Sunday=0) using the locale's alternative
 45211 numeric symbols.

45212 %OW Replaced by the week number of the year (Monday as the first day of the week) using
 45213 the locale's alternative numeric symbols.

45214 %OY Replaced by the year (offset from %C) using the locale's alternative numeric symbols.

45215 %g, %G, and %V give values according to the ISO 8601: 2000 standard week-based year. In this
 45216 system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th,
 45217 which is also the week that includes the first Thursday of the year, and is also the first week that
 45218 contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the
 45219 preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January
 45220 1999, %G is replaced by 1998 and %V is replaced by 53. If December 29th, 30th, or 31st is a
 45221 Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday
 45222 30th December 1997, %G is replaced by 1998 and %V is replaced by 01.

45223 If a conversion specifier is not one of the above, the behavior is undefined.

45224 RETURN VALUE

45225 If the total number of resulting bytes including the terminating null byte is not more than
 45226 maxsize, *strftime()* shall return the number of bytes placed into the array pointed to by *s*, not
 45227 including the terminating null byte. Otherwise, 0 shall be returned and the contents of the array
 45228 are unspecified.

45229 ERRORS

45230 No errors are defined.

45231 EXAMPLES

45232 Getting a Localized Date String

45233 The following example first sets the locale to the user's default. The locale information will be
 45234 used in the *nl_langinfo()* and *strftime()* functions. The *nl_langinfo()* function returns the localized
 45235 date string which specifies how the date is laid out. The *strftime()* function takes this information
 45236 and, using the **tm** structure for values, places the date and time information into *datestring*.

```
45237 #include <time.h>
45238 #include <locale.h>
45239 #include <langinfo.h>
45240 ...
45241 struct tm *tm;
45242 char datestring[256];
45243 ...
45244 setlocale (LC_ALL, " ");
45245 ...
45246 strftime (datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);
45247 ...
```

45248 APPLICATION USAGE

45249 The range of values for %S is [00,60] rather than [00,59] to allow for the occasional leap second.

45250 Some of the conversion specifications are duplicates of others. They are included for
 45251 compatibility with *nl_ctime()* and *nl_ascctime()*, which were published in Issue 2.

45252 Applications should use %Y (4-digit years) in preference to %y (2-digit years).

45253 In the C locale, the E and O modifiers are ignored and the replacement strings for the following
 45254 specifiers are:

45255	%a	The first three characters of %A.
45256	%A	One of Sunday, Monday, ..., Saturday.
45257	%b	The first three characters of %B.
45258	%B	One of January, February, ..., December.
45259	%c	Equivalent to %a %b %e %T %Y.
45260	%p	One of AM or PM.
45261	%r	Equivalent to %I:%M:%S %p.
45262	%x	Equivalent to %m/%d/%y.
45263	%X	Equivalent to %T.
45264	%Z	Implementation-defined.

45265 RATIONALE

45266 None.

45267 FUTURE DIRECTIONS

45268 None.

45269 SEE ALSO

45270 *asctime()*, *clock()*, *ctime()*, *difftime()*, *getdate()*, *gmtime()*, *localtime()*, *mktme()*, *strptime()*, *time()*,
45271 *tzset()*, *utime()*, Base Definitions volume of IEEE Std 1003.1-2001, Section 7.3.5, LC_TIME,
45272 <time.h>

45273 CHANGE HISTORY

45274 First released in Issue 3.

45275 Issue 5

45276 The description of %OV is changed to be consistent with %V and defines Monday as the first day
45277 of the week.

45278 The description of %OY is clarified.

45279 Issue 6

45280 Extensions beyond the ISO C standard are marked.

45281 The Open Group Corrigendum U033/8 is applied. The %V conversion specifier is changed from
45282 “Otherwise, it is week 53 of the previous year, and the next week is week 1” to “Otherwise, it is
45283 the last week of the previous year, and the next week is week 1”.

45284 The following new requirements on POSIX implementations derive from alignment with the
45285 Single UNIX Specification:

- The %C, %D, %e, %h, %n, %r, %R, %t, and %T conversion specifiers are added.
- The modified conversion specifiers are added for consistency with the ISO POSIX-2 standard
45288 *date* utility.

45289 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *strftime()* prototype is updated.
- The DESCRIPTION is extensively revised.
- The %z conversion specifier is added.

45293 A new example is added.

45294

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/60 is applied.

45295 NAME

45296 *strlen* — get string length

45297 SYNOPSIS

```
45298        #include <string.h>
45299        size_t strlen(const char *s);
```

45300 DESCRIPTION

45301 CX The functionality described on this reference page is aligned with the ISO C standard. Any
45302 conflict between the requirements described here and the ISO C standard is unintentional. This
45303 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

45304 The *strlen()* function shall compute the number of bytes in the string to which *s* points, not
45305 including the terminating null byte.

45306 RETURN VALUE

45307 The *strlen()* function shall return the length of *s*; no return value shall be reserved to indicate an
45308 error.

45309 ERRORS

45310 No errors are defined.

45311 EXAMPLES**45312 Getting String Lengths**

45313 The following example sets the maximum length of *key* and *data* by using *strlen()* to get the
45314 lengths of those strings.

```
45315        #include <string.h>
45316        ...
45317        struct element {
45318            char *key;
45319            char *data;
45320        };
45321        ...
45322        char *key, *data;
45323        int len;
45324        *keylength = *datalength = 0;
45325        ...
45326        if ((len = strlen(key)) > *keylength)
45327            *keylength = len;
45328        if ((len = strlen(data)) > *datalength)
45329            *datalength = len;
45330        ...
```

45331 APPLICATION USAGE

45332 None.

45333 RATIONALE

45334 None.

45335 FUTURE DIRECTIONS

45336 None.

45337 SEE ALSO

45338 The Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

45339 CHANGE HISTORY

45340 First released in Issue 1. Derived from Issue 1 of the SVID.

45341 Issue 5

45342 The RETURN VALUE section is updated to indicate that *strlen()* returns the length of *s*, and not *s* itself as was previously stated.
45343

45344 NAME

45345 strncasecmp — case-insensitive string comparison

45346 SYNOPSIS

45347 XSI #include <strings.h>

45348 int strncasecmp(const char *s1, const char *s2, size_t n);

45349

45350 DESCRIPTION

45351 Refer to *strcasecmp()*.

45352 NAME

45353 **strncat** — concatenate a string with part of another

45354 SYNOPSIS

45355

```
#include <string.h>
```

45356

```
char *strncat(char *restrict s1, const char *restrict s2, size_t n);
```

45357 DESCRIPTION

45358 CX 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.

45361 The *strncat()* function shall append not more than *n* bytes (a null byte and bytes that follow it are not appended) from the array pointed to by *s2* to the end of the string pointed to by *s1*. The initial byte of *s2* overwrites the null byte at the end of *s1*. A terminating null byte is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

45366 RETURN VALUE

45367 The *strncat()* function shall return *s1*; no return value shall be reserved to indicate an error.

45368 ERRORS

45369 No errors are defined.

45370 EXAMPLES

45371 None.

45372 APPLICATION USAGE

45373 None.

45374 RATIONALE

45375 None.

45376 FUTURE DIRECTIONS

45377 None.

45378 SEE ALSO

45379 *strcat()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**string.h**>

45380 CHANGE HISTORY

45381 First released in Issue 1. Derived from Issue 1 of the SVID.

45382 Issue 6

45383 The *strncat()* prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

45384 NAME

45385 `strcmp` — compare part of two strings

45386 SYNOPSIS

45387 `#include <string.h>`
45388 `int strcmp(const char *s1, const char *s2, size_t n);`

45389 DESCRIPTION

45390 CX 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.

45393 The `strcmp()` function shall compare not more than *n* bytes (bytes that follow a null byte are not compared) from the array pointed to by *s1* to the array pointed to by *s2*.

45395 The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings being compared.

45398 RETURN VALUE

45399 Upon successful completion, `strcmp()` shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by *s1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *s2* respectively.

45402 ERRORS

45403 No errors are defined.

45404 EXAMPLES

45405 None.

45406 APPLICATION USAGE

45407 None.

45408 RATIONALE

45409 None.

45410 FUTURE DIRECTIONS

45411 None.

45412 SEE ALSO

45413 `strcmp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<string.h>`

45414 CHANGE HISTORY

45415 First released in Issue 1. Derived from Issue 1 of the SVID.

45416 Issue 6

45417 Extensions beyond the ISO C standard are marked.

45418 NAME

45419 *strncpy* — copy part of a string

45420 SYNOPSIS

45421 #include <string.h>

45422 char *strncpy(char *restrict *s1*, const char *restrict *s2*, size_t *n*);

45423 DESCRIPTION

45424 CX 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.

45427 The *strncpy()* function shall copy not more than *n* bytes (bytes that follow a null byte are not copied) from the array pointed to by *s2* to the array pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

45430 If the array pointed to by *s2* is a string that is shorter than *n* bytes, null bytes shall be appended to the copy in the array pointed to by *s1*, until *n* bytes in all are written.

45432 RETURN VALUE

45433 The *strncpy()* function shall return *s1*; no return value is reserved to indicate an error.

45434 ERRORS

45435 No errors are defined.

45436 EXAMPLES

45437 None.

45438 APPLICATION USAGE

45439 Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

45441 If there is no null byte in the first *n* bytes of the array pointed to by *s2*, the result is not null-terminated.

45443 RATIONALE

45444 None.

45445 FUTURE DIRECTIONS

45446 None.

45447 SEE ALSO

45448 *strcpy()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

45449 CHANGE HISTORY

45450 First released in Issue 1. Derived from Issue 1 of the SVID.

45451 Issue 6

45452 The *strncpy()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

45453 NAME

45454 `strpbrk` — scan a string for a byte

45455 SYNOPSIS

45456 `#include <string.h>`

45457 `char *strpbrk(const char *s1, const char *s2);`

45458 DESCRIPTION

45459 CX 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.

45462 The `strpbrk()` function shall locate the first occurrence in the string pointed to by *s1* of any byte from the string pointed to by *s2*.

45464 RETURN VALUE

45465 Upon successful completion, `strpbrk()` shall return a pointer to the byte or a null pointer if no byte from *s2* occurs in *s1*.

45467 ERRORS

45468 No errors are defined.

45469 EXAMPLES

45470 None.

45471 APPLICATION USAGE

45472 None.

45473 RATIONALE

45474 None.

45475 FUTURE DIRECTIONS

45476 None.

45477 SEE ALSO

45478 `strchr()`, `strrchr()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<string.h>`

45479 CHANGE HISTORY

45480 First released in Issue 1. Derived from Issue 1 of the SVID.

45481 NAME

45482 strptime — date and time conversion

45483 SYNOPSIS

45484 XSI #include <time.h>

```
45485     char *strptime(const char *restrict buf, const char *restrict format,
45486             struct tm *restrict tm);
```

45487

45488 DESCRIPTION

45489 The *strptime()* function shall convert the character string pointed to by *buf* to values which are
45490 stored in the *tm* structure pointed to by *tm*, using the format specified by *format*.

45491 The *format* is composed of zero or more directives. Each directive is composed of one of the
45492 following: one or more white-space characters (as specified by *isspace()*); an ordinary character
45493 (neither '%' nor a white-space character); or a conversion specification. Each conversion
45494 specification is composed of a '%' character followed by a conversion character which specifies
45495 the replacement required. The application shall ensure that there is white-space or other non-
45496 alphanumeric characters between any two conversion specifications. The following conversion
45497 specifications are supported:

45498 %a	The day of the week, using the locale's weekday names; either the abbreviated or full 45499 name may be specified.
45500 %A	Equivalent to %a.
45501 %b	The month, using the locale's month names; either the abbreviated or full name may be 45502 specified.
45503 %B	Equivalent to %b.
45504 %c	Replaced by the locale's appropriate date and time representation.
45505 %C	The century number [00,99]; leading zeros are permitted but not required.
45506 %d	The day of the month [01,31]; leading zeros are permitted but not required.
45507 %D	The date as %m/%d/%y.
45508 %e	Equivalent to %d.
45509 %h	Equivalent to %b.
45510 %H	The hour (24-hour clock) [00,23]; leading zeros are permitted but not required.
45511 %I	The hour (12-hour clock) [01,12]; leading zeros are permitted but not required.
45512 %j	The day number of the year [001,366]; leading zeros are permitted but not required.
45513 %m	The month number [01,12]; leading zeros are permitted but not required.
45514 %M	The minute [00,59]; leading zeros are permitted but not required.
45515 %n	Any white space.
45516 %p	The locale's equivalent of a.m or p.m.
45517 %r	12-hour clock time using the AM/PM notation if <i>t_fmt_ampm</i> is not an empty string in 45518 the LC_TIME portion of the current locale; in the POSIX locale, this shall be equivalent 45519 to %I:%M:%S %p.
45520 %R	The time as %H:%M.

45521	%S	The seconds [00,60]; leading zeros are permitted but not required.
45522	%t	Any white space.
45523	%T	The time as %H:%M:%S.
45524	%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.
45525		
45526	%w	The weekday as a decimal number [0,6], with 0 representing Sunday; leading zeros are permitted but not required.
45527		
45528	%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.
45529		
45530	%x	The date, using the locale's date format.
45531	%X	The time, using the locale's time format.
45532	%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.
45533		
45534		
45535		
45536	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.)
45537		
45538		
45539	%Y	The year, including the century (for example, 1988).
45540	%%	Replaced by %.

Modified Conversion Specifiers

45542	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.
45543	
45544	
45545	
45546	%Ec The locale's alternative appropriate date and time representation.
45547	%Ec The name of the base year (period) in the locale's alternative representation.
45548	%Ex The locale's alternative date representation.
45549	%Ex The locale's alternative time representation.
45550	%Ey The offset from %Ec (year only) in the locale's alternative representation.
45551	%EY The full alternative year representation.
45552	%Od The day of the month using the locale's alternative numeric symbols; leading zeros are permitted but not required.
45553	
45554	%Oe Equivalent to %Od.
45555	%Oh The hour (24-hour clock) using the locale's alternative numeric symbols.
45556	%OI The hour (12-hour clock) using the locale's alternative numeric symbols.
45557	%Om The month using the locale's alternative numeric symbols.
45558	%Om The minutes using the locale's alternative numeric symbols.

45559	%OS	The seconds using the locale's alternative numeric symbols.	
45560	%OU	The week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.	
45561			
45562	%Ow	The number of the weekday (Sunday=0) using the locale's alternative numeric symbols.	
45563	%OW	The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.	
45564			
45565	%OY	The year (offset from %C) using the locale's alternative numeric symbols.	
45566		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.	
45567			
45568			
45569		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.	
45570			
45571			
45572			
45573		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.	
45574			
45575			
45576		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 buf such as month or weekday names. If no match is found, strftime() fails and no more characters are scanned.	
45577			
45578			
45579			
45580			
45581			
45582			

45583 RETURN VALUE

Upon successful completion, *strftime()* shall return a pointer to the character following the last character parsed. Otherwise, a null pointer shall be returned.

45586 ERRORS

45587 No errors are defined.

45588 EXAMPLES

45589	Convert a Data-Plus-Time String to Broken-Down Time and Then into Seconds	2
45590	The following example demonstrates the use of <i>strftime()</i> to convert a string into broken-down time. The broken-down time is then converted into seconds since the Epoch using <i>mktime()</i> .	2
45591		2
45592	#include <time.h>	2
45593	...	2
45594	struct tm tm;	2
45595	time_t t;	2
45596	if (strftime("6 Dec 2001 12:33:45", "%d %b %Y %H:%M:%S", &tm) == NULL)	2
45597	/* Handle error */;	2
45598	printf("year: %d; month: %d; day: %d;\n",	2
45599	tm.tm_year, tm.tm_mon, tm.tm_mday);	2
45600	printf("hour: %d; minute: %d; second: %d\n",	2
45601	tm.tm_hour, tm.tm_min, tm.tm_sec);	2

```

45602     printf("week day: %d; year day: %d\n", tm.tm_wday, tm.tm_yday);      2
45603     tm.tm_isdst = -1;          /* Not set by strftime(); tells mktime()      2
45604             to determine whether daylight saving time      2
45605             is in effect */      2
45606     t = mktime(&tm);      2
45607     if (t == -1)      2
45608         /* Handle error */;      2
45609     printf("seconds since the Epoch: %ld\n", (long) t);      2

```

45610 APPLICATION USAGE

45611 Several “equivalent to” formats and the special processing of white-space characters are
45612 provided in order to ease the use of identical *format* strings for *strftime()* and *strptime()*.

45613 Applications should use %Y (4-digit years) in preference to %y (2-digit years).

45614 It is unspecified whether multiple calls to *strptime()* using the same **tm** structure will update the
45615 current contents of the structure or overwrite all contents of the structure. Conforming
45616 applications should make a single call to *strptime()* with a format and all data needed to
45617 completely specify the date and time being converted.

45618 RATIONALE

45619 None.

45620 FUTURE DIRECTIONS

45621 The *strptime()* function is expected to be mandatory in the next version of this volume of
45622 IEEE Std 1003.1-2001.

45623 SEE ALSO

45624 *scanf()*, *strftime()*, *time()*, the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

45625 CHANGE HISTORY

45626 First released in Issue 4.

45627 Issue 5

45628 Moved from ENHANCED I18N to BASE.

45629 The [ENOSYS] error is removed.

45630 The exact meaning of the %y and %Oy specifiers is clarified in the DESCRIPTION.

45631 Issue 6

45632 The Open Group Corrigendum U033/5 is applied. The %r specifier description is reworded.

45633 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

45634 The **restrict** keyword is added to the *strptime()* prototype for alignment with the
45635 ISO/IEC 9899:1999 standard.

45636 The Open Group Corrigendum U047/2 is applied.

45637 The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion
45638 specification” for consistency with *strftime()*.

45639 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/133 is applied, adding the example to the
45640 EXAMPLES section. 2

45641 NAME

45642 *strrchr* — string scanning operation

45643 SYNOPSIS

```
45644        #include <string.h>
45645        char *strrchr(const char *s, int c);
```

45646 DESCRIPTION

45647 CX 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.

45650 The *strrchr()* function shall locate the last occurrence of *c* (converted to a **char**) in the string pointed to by *s*. The terminating null byte is considered to be part of the string.

45652 RETURN VALUE

45653 Upon successful completion, *strrchr()* shall return a pointer to the byte or a null pointer if *c* does not occur in the string.

45655 ERRORS

45656 No errors are defined.

45657 EXAMPLES**45658 Finding the Base Name of a File**

45659 The following example uses *strrchr()* to get a pointer to the base name of a file. The *strrchr()* function searches backwards through the name of the file to find the last ‘/’ character in *name*. This pointer (plus one) will point to the base name of the file.

```
45662        #include <string.h>
45663        ...
45664        const char *name;
45665        char *basename;
45666        ...
45667        basename = strrchr(name, '/') + 1;
45668        ...
```

45669 APPLICATION USAGE

45670 None.

45671 RATIONALE

45672 None.

45673 FUTURE DIRECTIONS

45674 None.

45675 SEE ALSO

45676 *strchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

45677 CHANGE HISTORY

45678 First released in Issue 1. Derived from Issue 1 of the SVID.

45679 NAME

45680 strspn — get length of a substring

45681 SYNOPSIS

```
45682 #include <string.h>
45683 size_t strspn(const char *s1, const char *s2);
```

45684 DESCRIPTION

45685 CX 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.

45688 The *strspn()* function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by *s1* which consists entirely of bytes from the string pointed to by *s2*.

45690 RETURN VALUE

45691 The *strspn()* function shall return the length of *s1*; no return value is reserved to indicate an error.

45693 ERRORS

45694 No errors are defined.

45695 EXAMPLES

45696 None.

45697 APPLICATION USAGE

45698 None.

45699 RATIONALE

45700 None.

45701 FUTURE DIRECTIONS

45702 None.

45703 SEE ALSO

45704 *strcspn()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

45705 CHANGE HISTORY

45706 First released in Issue 1. Derived from Issue 1 of the SVID.

45707 Issue 5

45708 The RETURN VALUE section is updated to indicate that *strspn()* returns the length of *s*, and not *s* itself as was previously stated.

45710 NAME

45711 strstr — find a substring

45712 SYNOPSIS

45713 #include <string.h>

45714 char *strstr(const char *s1, const char *s2);

45715 DESCRIPTION

45716 CX 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.

45719 The *strstr()* function shall locate the first occurrence in the string pointed to by *s1* of the sequence of bytes (excluding the terminating null byte) in the string pointed to by *s2*.

45721 RETURN VALUE

45722 Upon successful completion, *strstr()* shall return a pointer to the located string or a null pointer if the string is not found.

45724 If *s2* points to a string with zero length, the function shall return *s1*.

45725 ERRORS

45726 No errors are defined.

45727 EXAMPLES

45728 None.

45729 APPLICATION USAGE

45730 None.

45731 RATIONALE

45732 None.

45733 FUTURE DIRECTIONS

45734 None.

45735 SEE ALSO

45736 *strchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

45737 CHANGE HISTORY

45738 First released in Issue 3. Included for alignment with the ANSI C standard.

45739 NAME

45740 strtod, strtodf, strtold — convert a string to a double-precision number

45741 SYNOPSIS

```
45742     #include <stdlib.h>
45743
45744     double strtod(const char *restrict nptr, char **restrict endptr);
45745     float strtodf(const char *restrict nptr, char **restrict endptr);
45746     long double strtold(const char *restrict nptr, char **restrict endptr);
```

45746 DESCRIPTION

45747 CX 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.

45750 These functions shall convert the initial portion of the string pointed to by *nptr* to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts:

- 45753 1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace()*)
- 45754 2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
- 45755 3. A final string of one or more unrecognized characters, including the terminating null byte of the input string

45757 Then they shall attempt to convert the subject sequence to a floating-point number, and return the result.

45759 The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- 45761 • A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part
- 45763 • A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part
- 45765 • One of INF or INFINITY, ignoring case
- 45766 • One of NAN or NAN(*n-char-sequence*_{opt}), ignoring case in the NAN part, where:

```
45767     n-char-sequence:
45768         digit
45769         nondigit
45770         n-char-sequence digit
45771         n-char-sequence nondigit
```

45772 The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

45775 If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) shall be interpreted as a floating constant of the C language, except that the radix character shall be used in place of a period, and that if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence shall be interpreted as negated. A character sequence INF or INFINITY shall be interpreted as an

infinity, if representable in the return type, else as if it were a floating constant that is too large for the range of the return type. A character sequence NAN or NAN(*n-char-sequence_{opt}*) shall be interpreted as a quiet NaN, if supported in the return type, else as if it were a subject sequence part that does not have the expected form; the meaning of the *n*-char sequences is implementation-defined. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the value resulting from the conversion is correctly rounded.

The radix character is defined in the program's locale (category LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period ('.').

In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The *strtod()* function shall not change the setting of *errno* if successful.

Since 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set *errno* to 0, then call *strtod()*, *strtof()*, or *strtold()*, then check *errno*.

45802 RETURN VALUE

Upon successful completion, these functions shall return the converted value. If no conversion could be performed, 0 shall be returned, and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, ±HUGE_VAL, ±HUGE_VALF, or ±HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to [ERANGE].

If the correct value would cause an underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type shall be returned and *errno* set to [ERANGE].

45811 ERRORS

These functions shall fail if:

45813 CX [ERANGE] The value to be returned would cause overflow or underflow.

These functions may fail if:

45815 CX [EINVAL] No conversion could be performed.

45816 EXAMPLES

45817 None.

45818 APPLICATION USAGE

If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

If the subject sequence has the decimal form and at most DECIMAL_DIG (defined in <float.h>) significant digits, the result should be correctly rounded. If the subject sequence *D* has the decimal form and more than DECIMAL_DIG significant digits, consider the two bounding, 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:

```
45834 int
45835 what_kind_of_number (char *s)
45836 {
45837     char *endp;
45838     double d;
45839     long l;
45840
45841     d = strtod(s, &endp);
45842     if (s != endp && *endp == '\0')
45843         printf("It's a float with value %g\n", d);
45844     else
45845         {
45846             l = strtol(s, &endp, 0);
45847             if (s != endp && *endp == '\0')
45848                 printf("It's an integer with value %ld\n", l);
45849             else
45850                 return 1;
45851         }
45852     return 0;
45853 }
```

If the function is called with:

```
45854 what_kind_of_number ("0x10")
```

an ISO/IEC 9899:1990 standard-compliant library will result in the function printing:

```
45855 It's an integer with value 16
```

With the ISO/IEC 9899:1999 standard, the result is:

```
45858 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.

45861 RATIONALE

45862 None.

45863 FUTURE DIRECTIONS

45864 None.

45865 SEE ALSO

45866 *isspace()*, *localeconv()*, *scanf()*, *setlocale()*, *strtol()*, the Base Definitions volume of
45867 IEEE Std 1003.1-2001, Chapter 7, Locale, *<float.h>*, *<stdlib.h>*

45868 CHANGE HISTORY

45869 First released in Issue 1. Derived from Issue 1 of the SVID.

45870 Issue 5

45871 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

45872 Issue 6

45873 Extensions beyond the ISO C standard are marked.

45874 The following new requirements on POSIX implementations derive from alignment with the
45875 Single UNIX Specification:

- 45876 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
45877 added if no conversion could be performed.

45878 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 45879 • The *strtod()* function is updated.
- 45880 • The *strtof()* and *strtold()* functions are added.
- 45881 • The DESCRIPTION is extensively revised.

45882 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

45883 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/61 is applied, correcting the second 1
45884 paragraph in the RETURN VALUE section. This change clarifies the sign of the return value. 1

45885 NAME

45886 strtoimax, strtoumax — convert string to integer type

45887 SYNOPSIS

```
45888       #include <inttypes.h>
45889
45890       intmax_t strtoimax(const char *restrict nptr, char **restrict endptr,
45891                    int base);
45892       uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr,
45893                    int base);
```

45893 DESCRIPTION

45894 CX 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.

45897 These functions shall be equivalent to the *strtol()*, *strtoll()*, *strtoul()*, and *strtoull()* functions, except that the initial portion of the string shall be converted to **intmax_t** and **uintmax_t** representation, respectively.

45900 RETURN VALUE

45901 These functions shall return the converted value, if any.

45902 If no conversion could be performed, zero shall be returned.

45903 If the correct value is outside the range of representable values, {INTMAX_MAX},
45904 {INTMAX_MIN}, or {UINTMAX_MAX} shall be returned (according to the return type and sign
45905 of the value, if any), and *errno* shall be set to [ERANGE].

45906 ERRORS

45907 These functions shall fail if:

45908 [ERANGE] The value to be returned is not representable.

45909 These functions may fail if:

45910 [EINVAL] The value of *base* is not supported.

45911 EXAMPLES

45912 None.

45913 APPLICATION USAGE

45914 None.

45915 RATIONALE

45916 None.

45917 FUTURE DIRECTIONS

45918 None.

45919 SEE ALSO

45920 *strtol()*, *strtoul()*, the Base Definitions volume of IEEE Std 1003.1-2001, <inttypes.h>

45921 CHANGE HISTORY

45922 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

45923 NAME

45924 strtok, strtok_r — split string into tokens

45925 SYNOPSIS

```
45926       #include <string.h>
45927       char *strtok(char *restrict s1, const char *restrict s2);
45928 TSF      char *strtok_r(char *restrict s, const char *restrict sep,
45929            char **restrict lasts);
```

45931 DESCRIPTION

45932 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.

45935 A sequence of calls to *strtok()* breaks the string pointed to by *s1* into a sequence of tokens, each
45936 of which is delimited by a byte from the string pointed to by *s2*. The first call in the sequence has
45937 *s1* as its first argument, and is followed by calls with a null pointer as their first argument. The
45938 separator string pointed to by *s2* may be different from call to call.

45939 The first call in the sequence searches the string pointed to by *s1* for the first byte that is *not*
45940 contained in the current separator string pointed to by *s2*. If no such byte is found, then there
45941 are no tokens in the string pointed to by *s1* and *strtok()* shall return a null pointer. If such a byte
45942 is found, it is the start of the first token.

45943 The *strtok()* function then searches from there for a byte that *is* contained in the current
45944 separator string. If no such byte is found, the current token extends to the end of the string
45945 pointed to by *s1*, and subsequent searches for a token shall return a null pointer. If such a byte is
45946 found, it is overwritten by a null byte, which terminates the current token. The *strtok()* function
45947 saves a pointer to the following byte, from which the next search for a token shall start.

45948 Each subsequent call, with a null pointer as the value of the first argument, starts searching from
45949 the saved pointer and behaves as described above.

45950 The implementation shall behave as if no function defined in this volume of
45951 IEEE Std 1003.1-2001 calls *strtok()*.

45952 CX The *strtok()* function need not be reentrant. A function that is not required to be reentrant is not
45953 required to be thread-safe.

45954 TSF The *strtok_r()* function considers the null-terminated string *s* as a sequence of zero or more text
45955 tokens separated by spans of one or more characters from the separator string *sep*. The
45956 argument *lasts* points to a user-provided pointer which points to stored information necessary
45957 for *strtok_r()* to continue scanning the same string.

45958 In the first call to *strtok_r()*, *s* points to a null-terminated string, *sep* to a null-terminated string of
45959 separator characters, and the value pointed to by *lasts* is ignored. The *strtok_r()* function shall
45960 return a pointer to the first character of the first token, write a null character into *s* immediately
45961 following the returned token, and update the pointer to which *lasts* points.

45962 In subsequent calls, *s* is a NULL pointer and *lasts* shall be unchanged from the previous call so
45963 that subsequent calls shall move through the string *s*, returning successive tokens until no
45964 tokens remain. The separator string *sep* may be different from call to call. When no token
45965 remains in *s*, a NULL pointer shall be returned.

45966 RETURN VALUE

45967 Upon successful completion, *strtok()* shall return a pointer to the first byte of a token. Otherwise,
45968 if there is no token, *strtok()* shall return a null pointer.

45969 TSF The *strtok_r()* function shall return a pointer to the token found, or a NULL pointer when no
45970 token is found.

45971 ERRORS

45972 No errors are defined.

45973 EXAMPLES**45974 Searching for Word Separators**

45975 The following example searches for tokens separated by <space>s.

```
45976 #include <string.h>
45977 ...
45978 char *token;
45979 char *line = "LINE TO BE SEPARATED";
45980 char *search = " ";
45981 /* Token will point to "LINE". */
45982 token = strtok(line, search);
45983 /* Token will point to "TO". */
45984 token = strtok(NULL, search);
```

45985 Breaking a Line

45986 The following example uses *strtok()* to break a line into two character strings separated by any
45987 combination of <space>s, <tab>s, or <newline>s.

```
45988 #include <string.h>
45989 ...
45990 struct element {
45991     char *key;
45992     char *data;
45993 };
45994 ...
45995 char line[LINE_MAX];
45996 char *key, *data;
45997 ...
45998 key = strtok(line, " \n");
45999 data = strtok(NULL, " \n");
46000 ...
```

46001 APPLICATION USAGE

46002 The *strtok_r()* function is thread-safe and stores its state in a user-supplied buffer instead of
46003 possibly using a static data area that may be overwritten by an unrelated call from another
46004 thread.

46005 RATIONALE

46006 The *strtok()* function searches for a separator string within a larger string. It returns a pointer to
46007 the last substring between separator strings. This function uses static storage to keep track of
46008 the current string position between calls. The new function, *strtok_r()*, takes an additional
46009 argument, *lasts*, to keep track of the current position in the string.

46010 FUTURE DIRECTIONS

46011 None.

46012 SEE ALSO

46013 The Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

46014 CHANGE HISTORY

46015 First released in Issue 1. Derived from Issue 1 of the SVID.

46016 Issue 5

46017 The *strtok_r()* function is included for alignment with the POSIX Threads Extension.

46018 A note indicating that the *strtok()* function need not be reentrant is added to the DESCRIPTION.

46019 Issue 6

46020 Extensions beyond the ISO C standard are marked.

46021 The *strtok_r()* function is marked as part of the Thread-Safe Functions option.

46022 In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

46023 The APPLICATION USAGE section is updated to include a note on the thread-safe function and

46024 its avoidance of possibly using a static data area.

46025 The **restrict** keyword is added to the *strtok()* and *strtok_r()* prototypes for alignment with the

46026 ISO/IEC 9899:1999 standard.

46027 NAME

46028 `strtol, strtoll — convert a string to a long integer`

46029 SYNOPSIS

```
46030     #include <stdlib.h>
46031     long strtol(const char *restrict str, char **restrict endptr, int base);
46032     long long strtoll(const char *restrict str, char **restrict endptr,
46033                           int base)
```

46034 DESCRIPTION

46035 CX 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.

46038 These functions shall convert the initial portion of the string pointed to by *str* to a type **long** and **long long** representation, respectively. First, they decompose the input string into three parts:

- 46040 1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*())
- 46041 2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
- 46043 3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

46045 Then they shall attempt to convert the subject sequence to an integer, and return the result.

46046 If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

46052 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a '+' or '-' sign. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

46058 The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence shall contain no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

46062 If the subject sequence has the expected form and the value of *base* is 0, the sequence of characters starting with the first digit shall be interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

46069 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

46071 If the subject sequence is empty or does not have the expected form, no conversion is performed;
46072 the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null
46073 pointer.

46074 CX The *strtol()* function shall not change the setting of *errno* if successful.

46075 Since 0, {LONG_MIN} or {LLONG_MIN}, and {LONG_MAX} or {LLONG_MAX} are returned on
46076 error and are also valid returns on success, an application wishing to check for error situations
46077 should set *errno* to 0, then call *strtol()* or *strtoll()*, then check *errno*.

46078 RETURN VALUE

46079 Upon successful completion, these functions shall return the converted value, if any. If no
46080 CX conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL].

46081 If the correct value is outside the range of representable values, {LONG_MIN}, {LONG_MAX},
46082 {LLONG_MIN}, or {LLONG_MAX} shall be returned (according to the sign of the value), and
46083 *errno* set to [ERANGE].

46084 ERRORS

46085 These functions shall fail if:

46086 [ERANGE] The value to be returned is not representable.

46087 These functions may fail if:

46088 CX [EINVAL] The value of *base* is not supported.

46089 EXAMPLES

46090 None.

46091 APPLICATION USAGE

46092 None.

46093 RATIONALE

46094 None.

46095 FUTURE DIRECTIONS

46096 None.

46097 SEE ALSO

46098 *isalpha()*, *scanf()*, *strtod()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

46099 CHANGE HISTORY

46100 First released in Issue 1. Derived from Issue 1 of the SVID.

46101 Issue 5

46102 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

46103 Issue 6

46104 Extensions beyond the ISO C standard are marked.

46105 The following new requirements on POSIX implementations derive from alignment with the
46106 Single UNIX Specification:

- 46107 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
46108 added if no conversion could be performed.

46109 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 46110 • The *strtol()* prototype is updated.
46111 • The *strtoll()* function is added.

46112 NAME

46113 `strtold` — convert a string to a double-precision number

46114 SYNOPSIS

46115 `#include <stdlib.h>`

46116 `long double strtold(const char *restrict nptr, char **restrict endptr);`

46117 DESCRIPTION

46118 Refer to `strtod()`.

46119 NAME

46120 **strtoll** — convert a string to a long integer

46121 SYNOPSIS

46122 #include <stdlib.h>

46123 long long strtoll(const char *restrict *str*, char **restrict *endptr*,
46124 int *base*);

46125 DESCRIPTION

46126 Refer to *strtol()*.

46127 NAME

46128 strtoul, strtoull — convert a string to an unsigned long

46129 SYNOPSIS

```
46130     #include <stdlib.h>
46131
46132     unsigned long strtoul(const char *restrict str,
46133         char **restrict endptr, int base);
46134     unsigned long long strtoull(const char *restrict str,
46135         char **restrict endptr, int base);
```

46135 DESCRIPTION

46136 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 46137 conflict between the requirements described here and the ISO C standard is unintentional. This
 46138 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

46139 These functions shall convert the initial portion of the string pointed to by *str* to a type **unsigned**
 46140 **long** and **unsigned long long** representation, respectively. First, they decompose the input
 46141 string into three parts:

- 46142 1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace()*)
- 46143 2. A subject sequence interpreted as an integer represented in some radix determined by the
 value of *base*
- 46145 3. A final string of one or more unrecognized characters, including the terminating null byte
 of the input string

46147 Then they shall attempt to convert the subject sequence to an unsigned integer, and return the
 46148 result.

46149 If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant,
 46150 octal constant, or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A
 46151 decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An
 46152 octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to
 46153 '7' only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
 46154 decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

46155 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
 46156 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
 46157 by a '+' or '-' sign. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the
 46158 values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the
 46159 value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and
 46160 digits, following the sign if present.

46161 The subject sequence is defined as the longest initial subsequence of the input string, starting
 46162 with the first non-white-space character that is of the expected form. The subject sequence shall
 46163 contain no characters if the input string is empty or consists entirely of white-space characters,
 46164 or if the first non-white-space character is other than a sign or a permissible letter or digit.

46165 If the subject sequence has the expected form and the value of *base* is 0, the sequence of
 46166 characters starting with the first digit shall be interpreted as an integer constant. If the subject
 46167 sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the
 46168 base for conversion, ascribing to each letter its value as given above. If the subject sequence
 46169 begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to
 46170 the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null
 46171 pointer.

46172 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

46174 If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of *str* shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

46177 CX The *strtoul()* function shall not change the setting of *errno* if successful.

46178 Since 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and are also valid returns
46179 on success, an application wishing to check for error situations should set *errno* to 0, then call
46180 *strtoul()* or *strtoull()*, then check *errno*.

46181 RETURN VALUE

46182 Upon successful completion, these functions shall return the converted value, if any. If no
46183 CX conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL]. If the
46184 correct value is outside the range of representable values, {ULONG_MAX} or {ULLONG_MAX}
46185 shall be returned and *errno* set to [ERANGE].

46186 ERRORS

46187 These functions shall fail if:

46188 CX [EINVAL] The value of *base* is not supported.

46189 [ERANGE] The value to be returned is not representable.

46190 These functions may fail if:

46191 CX [EINVAL] No conversion could be performed.

46192 EXAMPLES

46193 None.

46194 APPLICATION USAGE

46195 None.

46196 RATIONALE

46197 None.

46198 FUTURE DIRECTIONS

46199 None.

46200 SEE ALSO

46201 *isalpha()*, *scanf()*, *strtod()*, *strtol()*, the Base Definitions volume of IEEE Std 1003.1-2001,
46202 <stdlib.h>

46203 CHANGE HISTORY

46204 First released in Issue 4. Derived from the ANSI C standard.

46205 Issue 5

46206 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

46207 Issue 6

46208 Extensions beyond the ISO C standard are marked.

46209 The following new requirements on POSIX implementations derive from alignment with the
46210 Single UNIX Specification:

- The [EINVAL] error condition is added for when the value of *base* is not supported.

46212 In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
46213 added if no conversion could be performed.

46214

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

46215

- The *strtoul()* prototype is updated.
- The *strtoull()* function is added.

46216

46217 **NAME**46218 **strtoumax** — convert a string to an integer type46219 **SYNOPSIS**46220 #include <inttypes.h>
46221 uintmax_t strtoumax(const char *restrict *nptr*, char **restrict *endptr*,
46222 int *base*);46223 **DESCRIPTION**46224 Refer to *strtoimax()*.

46225 NAME

46226 strxfrm — string transformation

46227 SYNOPSIS

46228 #include <string.h>
46229
46229 size_t strxfrm(char *restrict s1, const char *restrict s2, size_t n);

46230 DESCRIPTION

46231 CX 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.

46234 The *strxfrm()* function shall transform the string pointed to by *s2* and place the resulting string into the array pointed to by *s1*. The transformation is such that if *strcmp()* is applied to two transformed strings, it shall return a value greater than, equal to, or less than 0, corresponding to the result of *strcoll()* applied to the same two original strings. No more than *n* bytes are placed into the resulting array pointed to by *s1*, including the terminating null byte. If *n* is 0, *s1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

46241 CX The *strxfrm()* function shall not change the setting of *errno* if successful.

46242 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 *strxfrm()*, then check *errno*.

46244 RETURN VALUE

46245 Upon successful completion, *strxfrm()* shall return the length of the transformed string (not including the terminating null byte). If the value returned is *n* or more, the contents of the array pointed to by *s1* are unspecified.

46248 CX On error, *strxfrm()* may set *errno* but no return value is reserved to indicate an error.

46249 ERRORS

46250 The *strxfrm()* function may fail if:

46251 CX [EINVAL] The string pointed to by the *s2* argument contains characters outside the domain of the collating sequence.

46253 EXAMPLES

46254 None.

46255 APPLICATION USAGE

46256 The transformation function is such that two transformed strings can be ordered by *strcmp()* as appropriate to collating sequence information in the program's locale (category *LC_COLLATE*).

46258 The fact that when *n* is 0 *s1* is permitted to be a null pointer is useful to determine the size of the *s1* array prior to making the transformation.

46260 RATIONALE

46261 None.

46262 FUTURE DIRECTIONS

46263 None.

46264 SEE ALSO

46265 *strcmp()*, *strcoll()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

46266 CHANGE HISTORY

46267 First released in Issue 3. Included for alignment with the ISO C standard.

46268 Issue 5

46269 The DESCRIPTION is updated to indicate that *errno* does not change if the function is
46270 successful.

46271 Issue 6

46272 Extensions beyond the ISO C standard are marked.

46273 The following new requirements on POSIX implementations derive from alignment with the
46274 Single UNIX Specification:

- 46275 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
46276 added if no conversion could be performed.

46277 The *strxfrm()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

46278 NAME

46279 **swab** — swap bytes

46280 SYNOPSIS

46281 XSI #include <unistd.h>
46282 void swab(const void *restrict src, void *restrict dest,
46283 ssize_t nbytes);
46284

46285 DESCRIPTION

46286 The **swab()** function shall copy *nbytes* bytes, which are pointed to by *src*, to the object pointed to
46287 by *dest*, exchanging adjacent bytes. The *nbytes* argument should be even. If *nbytes* is odd, **swab()**
46288 copies and exchanges *nbytes*-1 bytes and the disposition of the last byte is unspecified. If
46289 copying takes place between objects that overlap, the behavior is undefined. If *nbytes* is
46290 negative, **swab()** does nothing.

46291 RETURN VALUE

46292 None.

46293 ERRORS

46294 No errors are defined.

46295 EXAMPLES

46296 None.

46297 APPLICATION USAGE

46298 None.

46299 RATIONALE

46300 None.

46301 FUTURE DIRECTIONS

46302 None.

46303 SEE ALSO

46304 The Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

46305 CHANGE HISTORY

46306 First released in Issue 1. Derived from Issue 1 of the SVID.

46307 Issue 6

46308 The **restrict** keyword is added to the **swab()** prototype for alignment with the
46309 ISO/IEC 9899:1999 standard.

46310 NAME

46311 **swapcontext** — swap user context

46312 SYNOPSIS

46313 OB XSI **#include <ucontext.h>**

2

46314 **int swapcontext(ucontext_t *restrict oucp,**

46315 **const ucontext_t *restrict ucp);**

46316

46317 DESCRIPTION

46318 Refer to *makecontext()*.

46319 NAME

46320 swprintf — print formatted wide-character output

46321 SYNOPSIS

46322 #include <stdio.h>

46323 #include <wchar.h>

46324 int swprintf(wchar_t *restrict ws, size_t n,

46325 const wchar_t *restrict format, ...);

46326 DESCRIPTION

46327 Refer to *fprintf()*.

46328 NAME

46329 **swscanf** — convert formatted wide-character input

46330 SYNOPSIS

```
46331        #include <stdio.h>
46332        #include <wchar.h>
46333        int swscanf(const wchar_t *restrict ws,
46334                    const wchar_t *restrict format, ... );
```

46335 DESCRIPTION

46336 Refer to *fwscanf()*.

46337 NAME

46338 symlink — make a symbolic link to a file

46339 SYNOPSIS

46340 #include <unistd.h>
46341 int symlink(const char *path1, const char *path2);

46342 DESCRIPTION

46343 The *symlink()* function shall create a symbolic link called *path2* that contains the string pointed to by *path1* (*path2* is the name of the symbolic link created, *path1* is the string contained in the symbolic link).

46346 The string pointed to by *path1* shall be treated only as a character string and shall not be validated as a pathname.

46348 If the *symlink()* function fails for any reason other than [EIO], any file named by *path2* shall be unaffected.

46350 RETURN VALUE

46351 Upon successful completion, *symlink()* shall return 0; otherwise, it shall return -1 and set *errno* to indicate the error.

46353 ERRORS

46354 The *symlink()* function shall fail if:

46355 [EACCES] Write permission is denied in the directory where the symbolic link is being created, or search permission is denied for a component of the path prefix of *path2*.

46358 [EEXIST] The *path2* argument names an existing file or symbolic link.

46359 [EIO] An I/O error occurs while reading from or writing to the file system.

46360 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path2* argument.

46362 [ENAMETOOLONG] The length of the *path2* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX} or the length of the *path1* argument is longer than {SYMLINK_MAX}.

46366 [ENOENT] A component of *path2* does not name an existing file or *path2* is an empty string.

46368 [ENOSPC] The directory in which the entry for the new symbolic link is being placed cannot be extended because no space is left on the file system containing the directory, or the new symbolic link cannot be created because no space is left on the file system which shall contain the link, or the file system is out of file-allocation resources.

46373 [ENOTDIR] A component of the path prefix of *path2* is not a directory.

46374 [EROFS] The new symbolic link would reside on a read-only file system.

46375 The *symlink()* function may fail if:

46376 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path2* argument.

46378 [ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the *path2*

46380 argument, the length of the substituted pathname string exceeded
46381 {PATH_MAX} bytes (including the terminating null byte), or the length of the
46382 string pointed to by *path1* exceeded {SYMLINK_MAX}.

46383 EXAMPLES

46384 None.

46385 APPLICATION USAGE

46386 Like a hard link, a symbolic link allows a file to have multiple logical names. The presence of a
46387 hard link guarantees the existence of a file, even after the original name has been removed. A
46388 symbolic link provides no such assurance; in fact, the file named by the *path1* argument need not
46389 exist when the link is created. A symbolic link can cross file system boundaries.

46390 Normal permission checks are made on each component of the symbolic link pathname during
46391 its resolution.

46392 RATIONALE

46393 Since IEEE Std 1003.1-2001 does not require any association of file times with symbolic links,
46394 there is no requirement that file times be updated by *symlink()*.

46395 FUTURE DIRECTIONS

46396 None.

46397 SEE ALSO

46398 *lchown()*, *link()*, *lstat()*, *open()*, *readlink()*, *unlink()*, the Base Definitions volume of
46399 IEEE Std 1003.1-2001, <unistd.h>

46400 CHANGE HISTORY

46401 First released in Issue 4, Version 2.

46402 Issue 5

46403 Moved from X/OPEN UNIX extension to BASE.

46404 Issue 6

46405 The following changes were made to align with the IEEE P1003.1a draft standard:

- 46406
- The DESCRIPTION text is updated.
 - The [ELOOP] optional error condition is added.

46408 NAME

46409 sync — schedule file system updates

46410 SYNOPSIS

46411 XSI #include <unistd.h>
46412 void sync(void);
46413

46414 DESCRIPTION

46415 The *sync()* function shall cause all information in memory that updates file systems to be
46416 scheduled for writing out to all file systems.

46417 The writing, although scheduled, is not necessarily complete upon return from *sync()*.

46418 RETURN VALUE

46419 The *sync()* function shall not return a value.

46420 ERRORS

46421 No errors are defined.

46422 EXAMPLES

46423 None.

46424 APPLICATION USAGE

46425 None.

46426 RATIONALE

46427 None.

46428 FUTURE DIRECTIONS

46429 None.

46430 SEE ALSO

46431 *fsync()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

46432 CHANGE HISTORY

46433 First released in Issue 4, Version 2.

46434 Issue 5

46435 Moved from X/OPEN UNIX extension to BASE.

46436 NAME

46437 sysconf — get configurable system variables

46438 SYNOPSIS

```
46439     #include <unistd.h>
46440     long sysconf(int name);
```

46441 DESCRIPTION

46442 The *sysconf()* function provides a method for the application to determine the current value of a
 46443 configurable system limit or option (*variable*). The implementation shall support all of the
 46444 variables listed in the following table and may support others.

46445 The *name* argument represents the system variable to be queried. The following table lists the
 46446 minimal set of system variables from <limits.h> or <unistd.h> that can be returned by *sysconf()*,
 46447 and the symbolic constants defined in <unistd.h> that are the corresponding values used for
 46448 *name*.

46450	Variable	Value of Name	
46451	{AIO_LISTIO_MAX}	_SC_AIO_LISTIO_MAX	1
46452	{AIO_MAX}	_SC_AIO_MAX	
46453	{AIO_PRIO_DELTA_MAX}	_SC_AIO_PRIO_DELTA_MAX	
46454	{ARG_MAX}	_SC_ARG_MAX	1
46455	{ATEXIT_MAX}	_SC_ATEXIT_MAX	1
46456	{BC_BASE_MAX}	_SC_BC_BASE_MAX	1
46457	{BC_DIM_MAX}	_SC_BC_DIM_MAX	
46458	{BC_SCALE_MAX}	_SC_BC_SCALE_MAX	
46459	{BC_STRING_MAX}	_SC_BC_STRING_MAX	
46460	{CHILD_MAX}	_SC_CHILD_MAX	
46461	Clock ticks/second	_SC_CLK_TCK	
46462	{COLL_WEIGHTS_MAX}	_SC_COLL_WEIGHTS_MAX	
46463	{DELAYTIMER_MAX}	_SC_DELAYTIMER_MAX	1
46464	{EXPR_NEST_MAX}	_SC_EXPR_NEST_MAX	1
46465	{HOST_NAME_MAX}	_SC_HOST_NAME_MAX	
46466	{IOV_MAX}	_SC_IOV_MAX	1
46467	{LINE_MAX}	_SC_LINE_MAX	1
46468	{LOGIN_NAME_MAX}	_SC_LOGIN_NAME_MAX	
46469	{NGROUPS_MAX}	_SC_NGROUPS_MAX	
46470	Maximum size of <i>getgrgid_r()</i> and 46471 <i>getgrnam_r()</i> data buffers	_SC_GETGR_R_SIZE_MAX	1
46472	Maximum size of <i>getpwuid_r()</i> and 46473 <i>getpwnam_r()</i> data buffers	_SC_GETPW_R_SIZE_MAX	
46474	{MQ_OPEN_MAX}	_SC_MQ_OPEN_MAX	1
46475	{MQ_PRIO_MAX}	_SC_MQ_PRIO_MAX	
46476	{OPEN_MAX}	_SC_OPEN_MAX	1
46477	_POSIX_ADVISORY_INFO	_SC_ADVISORY_INFO	1
46478	_POSIX_BARRIERS	_SC_BARRIERS	1
46479	_POSIX_ASYNCHRONOUS_IO	_SC_ASYNCHRONOUS_IO	1
46480	_POSIX_CLOCK_SELECTION	_SC_CLOCK_SELECTION	1
46481	_POSIX_CPUTIME	_SC_CPUTIME	2

	Variable	Value of Name	
46482			
46483			
46484	_POSIX_FSYNC	_SC_FSYNC	1
46485	_POSIX_IPV6	_SC_IPV6	1
46486	_POSIX_JOB_CONTROL	_SC_JOB_CONTROL	1
46487	_POSIX_MAPPED_FILES	_SC_MAPPED_FILES	1
46488	_POSIX_MEMLOCK	_SC_MEMLOCK	1
46489	_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE	1
46490	_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION	1
46491	_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING	1
46492	_POSIX_MONOTONIC_CLOCK	_SC_MONOTONIC_CLOCK	2
46493	_POSIX_PRIORITIZED_IO	_SC_PRIORITIZED_IO	1
46494	_POSIX_PRIORITY_SCHEDULING	_SC_PRIORITY_SCHEDULING	1
46495	_POSIX_RAW_SOCKETS	_SC_RAW_SOCKETS	1
46496	_POSIX_READER_WRITER_LOCKS	_SC_READER_WRITER_LOCKS	1
46497	_POSIX_REALTIME_SIGNALS	_SC_REALTIME_SIGNALS	1
46498	_POSIX_REGEXP	_SC_REGEXP	1
46499	_POSIX_SAVED_IDS	_SC_SAVED_IDS	
46500	_POSIX_SEMAPHORES	_SC_SEMAPHORES	1
46501	_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS	1
46502	_POSIX_SHELL	_SC_SHELL	1
46503	_POSIX_SPAWN	_SC_SPAWN	1
46504	_POSIX_SPIN_LOCKS	_SC_SPIN_LOCKS	1
46505	_POSIX_SPORADIC_SERVER	_SC_SPORADIC_SERVER	2
46506	_POSIX_SS_REPL_MAX	_SC_SS_REPL_MAX	1
46507	_POSIX_SYNCHRONIZED_IO	_SC_SYNCHRONIZED_IO	1
46508	_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR	1
46509	_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE	1
46510	_POSIX_THREAD_CPUTIME	_SC_THREAD_CPUTIME	1
46511	_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT	1
46512	_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT	1
46513	_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING	1
46514	_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED	1
46515	_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS	1
46516	_POSIX_THREAD_SPORADIC_SERVER	_SC_THREAD_SPORADIC_SERVER	1
46517	_POSIX_THREADS	_SC_THREADS	1
46518	_POSIX_TIMEOUTS	_SC_TIMEOUTS	1
46519	_POSIX_TIMERS	_SC_TIMERS	1
46520	_POSIX_TRACE	_SC_TRACE	1
46521	_POSIX_TRACE_EVENT_FILTER	_SC_TRACE_EVENT_FILTER	1
46522	_POSIX_TRACE_EVENT_NAME_MAX	_SC_TRACE_EVENT_NAME_MAX	2
46523	_POSIX_TRACE_INHERIT	_SC_TRACE_INHERIT	1
46524	_POSIX_TRACE_LOG	_SC_TRACE_LOG	1
46525	_POSIX_TRACE_NAME_MAX	_SC_TRACE_NAME_MAX	2
46526	_POSIX_TRACE_SYS_MAX	_SC_TRACE_SYS_MAX	2
46527	_POSIX_TRACE_USER_EVENT_MAX	_SC_TRACE_USER_EVENT_MAX	2
46528	_POSIX_TYPED_MEMORY_OBJECTS	_SC_TYPED_MEMORY_OBJECTS	1
46529	_POSIX_VERSION	_SC_VERSION	1
46530	_POSIX_V6_ILP32_OFF32	_SC_V6_ILP32_OFF32	

	Variable	Value of Name	
46531			
46532			
46533	_POSIX_V6_ILP32_OFFBIG	_SC_V6_ILP32_OFFBIG	1
46534	_POSIX_V6_LP64_OFF64	_SC_V6_LP64_OFF64	
46535	_POSIX_V6_LPBIG_OFFBIG	_SC_V6_LPBIG_OFFBIG	
46536	_POSIX2_C_BIND	_SC_2_C_BIND	
46537	_POSIX2_C_DEV	_SC_2_C_DEV	
46538	_POSIX2_CHAR_TERM	_SC_2_CHAR_TERM	2
46539	_POSIX2_FORT_DEV	_SC_2_FORT_DEV	
46540	_POSIX2_FORT_RUN	_SC_2_FORT_RUN	
46541	_POSIX2_LOCALEDEF	_SC_2_LOCALEDEF	
46542	_POSIX2_PBS	_SC_2_PBS	1
46543	_POSIX2_PBS_ACCOUNTING	_SC_2_PBS_ACCOUNTING	
46544	_POSIX2_PBS_CHECKPOINT	_SC_2_PBS_CHECKPOINT	1
46545	_POSIX2_PBS_LOCATE	_SC_2_PBS_LOCATE	1
46546	_POSIX2_PBS_MESSAGE	_SC_2_PBS_MESSAGE	
46547	_POSIX2_PBS_TRACK	_SC_2_PBS_TRACK	
46548	_POSIX2_SW_DEV	_SC_2_SW_DEV	1
46549	_POSIX2_UPE	_SC_2_UPE	
46550	_POSIX2_VERSION	_SC_2_VERSION	
46551	{PAGE_SIZE}	_SC_PAGE_SIZE	1
46552	{PAGESIZE}	_SC_PAGESIZE	1
46553	{PTHREAD_DESTRUCTOR_ITERATIONS}	_SC_THREAD_DESTRUCTOR_ITERATIONS	1
46554	{PTHREAD_KEYS_MAX}	_SC_THREAD_KEYS_MAX	
46555	{PTHREAD_STACK_MIN}	_SC_THREAD_STACK_MIN	
46556	{PTHREAD_THREADS_MAX}	_SC_THREAD_THREADS_MAX	
46557	{RE_DUP_MAX}	_SC_RE_DUP_MAX	1
46558	{RTSIG_MAX}	_SC_RTSIG_MAX	1
46559	{SEM_NSEMS_MAX}	_SC_SEM_NSEMS_MAX	1
46560	{SEM_VALUE_MAX}	_SC_SEM_VALUE_MAX	
46561	{SIGQUEUE_MAX}	_SC_SIGQUEUE_MAX	1
46562	{STREAM_MAX}	_SC_STREAM_MAX	1
46563	{SYMLOOP_MAX}	_SC_SYMLOOP_MAX	
46564	{TIMER_MAX}	_SC_TIMER_MAX	1
46565	{TTY_NAME_MAX}	_SC_TTY_NAME_MAX	1
46566	{TZNAME_MAX}	_SC_TZNAME_MAX	
46567	_XBS5_ILP32_OFF32 (LEGACY)	_SC_XBS5_ILP32_OFF32 (LEGACY)	1
46568	_XBS5_ILP32_OFFBIG (LEGACY)	_SC_XBS5_ILP32_OFFBIG (LEGACY)	
46569	_XBS5_LP64_OFF64 (LEGACY)	_SC_XBS5_LP64_OFF64 (LEGACY)	
46570	_XBS5_LPBIG_OFFBIG (LEGACY)	_SC_XBS5_LPBIG_OFFBIG (LEGACY)	
46571	_XOPEN_CRYPT	_SC_XOPEN_CRYPT	
46572	_XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N	
46573	_XOPEN_LEGACY	_SC_XOPEN_LEGACY	1
46574	_XOPEN_REALTIME	_SC_XOPEN_REALTIME	
46575	_XOPEN_REALTIME_THREADS	_SC_XOPEN_REALTIME_THREADS	
46576	_XOPEN_SHM	_SC_XOPEN_SHM	
46577	_XOPEN_STREAMS	_SC_XOPEN_STREAMS	1
46578	_XOPEN_UNIX	_SC_XOPEN_UNIX	1
46579	_XOPEN_VERSION	_SC_XOPEN_VERSION	2

46580 RETURN VALUE

46581 If *name* is an invalid value, *sysconf()* shall return -1 and set *errno* to indicate the error. If the
46582 variable corresponding to *name* has no limit, *sysconf()* shall return -1 without changing the value
46583 of *errno*. Note that indefinite limits do not imply infinite limits; see <**limits.h**>.

46584 Otherwise, *sysconf()* shall return the current variable value on the system. The value returned
46585 shall not be more restrictive than the corresponding value described to the application when it
46586 was compiled with the implementation's <**limits.h**> or <**unistd.h**>. The value shall not change
46587 XSI during the lifetime of the calling process, except that *sysconf(_SC_OPEN_MAX)* may return
46588 different values before and after a call to *setrlimit()* which changes the RLIMIT_NOFILE soft
46589 limit.

46590 If the variable corresponding to *name* is dependent on an unsupported option, the results are
46591 unspecified.

46592 ERRORS

46593 The *sysconf()* function shall fail if:

46594 [EINVAL] The value of the *name* argument is invalid.

46595 EXAMPLES

46596 None.

46597 APPLICATION USAGE

46598 As -1 is a permissible return value in a successful situation, an application wishing to check for
46599 error situations should set *errno* to 0, then call *sysconf()*, and, if it returns -1, check to see if *errno*
46600 is non-zero.

46601 Application writers should check whether an option, such as _POSIX_TRACE, is supported
46602 prior to obtaining and using values for related variables, such as _POSIX_TRACE_NAME_MAX.

46603 RATIONALE

46604 This functionality was added in response to requirements of application developers and of
46605 system vendors who deal with many international system configurations. It is closely related to
46606 *pathconf()* and *fpathconf()*.

46607 Although a conforming application can run on all systems by never demanding more resources
46608 than the minimum values published in this volume of IEEE Std 1003.1-2001, it is useful for that
46609 application to be able to use the actual value for the quantity of a resource available on any
46610 given system. To do this, the application makes use of the value of a symbolic constant in
46611 <**limits.h**> or <**unistd.h**>.

46612 However, once compiled, the application must still be able to cope if the amount of resource
46613 available is increased. To that end, an application may need a means of determining the quantity
46614 of a resource, or the presence of an option, at execution time.

46615 Two examples are offered:

- 46616 1. Applications may wish to act differently on systems with or without job control.
46617 Applications vendors who wish to distribute only a single binary package to all instances
46618 of a computer architecture would be forced to assume job control is never available if it
46619 were to rely solely on the <**unistd.h**> value published in this volume of
46620 IEEE Std 1003.1-2001.
- 46621 2. International applications vendors occasionally require knowledge of the number of clock
46622 ticks per second. Without these facilities, they would be required to either distribute their
46623 applications partially in source form or to have 50 Hz and 60 Hz versions for the various
46624 countries in which they operate.

46625 It is the knowledge that many applications are actually distributed widely in executable form
46626 that leads to this facility. If limited to the most restrictive values in the headers, such
46627 applications would have to be prepared to accept the most limited environments offered by the
46628 smallest microcomputers. Although this is entirely portable, there was a consensus that they
46629 should be able to take advantage of the facilities offered by large systems, without the
46630 restrictions associated with source and object distributions.

46631 During the discussions of this feature, it was pointed out that it is almost always possible for an
46632 application to discern what a value might be at runtime by suitably testing the various functions
46633 themselves. And, in any event, it could always be written to adequately deal with error returns
46634 from the various functions. In the end, it was felt that this imposed an unreasonable level of
46635 complication and sophistication on the application writer.

46636 This runtime facility is not meant to provide ever-changing values that applications have to
46637 check multiple times. The values are seen as changing no more frequently than once per system
46638 initialization, such as by a system administrator or operator with an automatic configuration
46639 program. This volume of IEEE Std 1003.1-2001 specifies that they shall not change within the
46640 lifetime of the process.

46641 Some values apply to the system overall and others vary at the file system or directory level. The
46642 latter are described in *pathconf()*.

46643 Note that all values returned must be expressible as integers. String values were considered, but
46644 the additional flexibility of this approach was rejected due to its added complexity of
46645 implementation and use.

46646 Some values, such as {PATH_MAX}, are sometimes so large that they must not be used to, say,
46647 allocate arrays. The *sysconf()* function returns a negative value to show that this symbolic
46648 constant is not even defined in this case.

46649 Similar to *pathconf()*, this permits the implementation not to have a limit. When one resource is
46650 infinite, returning an error indicating that some other resource limit has been reached is
46651 conforming behavior.

46652 FUTURE DIRECTIONS

46653 None.

46654 SEE ALSO

46655 *confstr()*, *pathconf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>,
46656 <unistd.h>, the Shell and Utilities volume of IEEE Std 1003.1-2001, *getconf*

46657 CHANGE HISTORY

46658 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

46659 Issue 5

46660 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
46661 Threads Extension.

46662 The _XBS_ variables and name values are added to the table of system variables in the
46663 DESCRIPTION. These are all marked EX.

46664 Issue 6

46665 The symbol CLK_TCK is obsolescent and removed. It is replaced with the phrase “clock ticks
46666 per second”.

46667 The symbol {PASS_MAX} is removed.

46668 The following changes were made to align with the IEEE P1003.1a draft standard:

- 46669 • Table entries are added for the following variables: _SC_REGEXP, _SC_SHELL,
 46670 _SC_REGEX_VERSION, _SC_SYMLOOP_MAX.

46671 The following *sysconf()* variables and their associated names are added for alignment with
 46672 IEEE Std 1003.1d-1999:

```
_POSIX_ADVISORY_INFO
_POSIX_CPUTIME
_POSIX_SPAWN
_POSIX_SPORADIC_SERVER
_POSIX_THREAD_CPUTIME
_POSIX_THREAD_SPORADIC_SERVER
_POSIX_TIMEOUTS
```

46680 The following changes are made to the DESCRIPTION for alignment with IEEE Std 1003.1j-2000:

- 46681 • A statement expressing the dependency of support for some system variables on
 46682 implementation options is added.
- 46683 • The following system variables are added:

```
_POSIX_BARRIERS
_POSIX_CLOCK_SELECTION
_POSIX_MONOTONIC_CLOCK
_POSIX_READER_WRITER_LOCKS
_POSIX_SPIN_LOCKS
_POSIX_TYPED_MEMORY_OBJECTS
```

46690 The following system variables are added for alignment with IEEE Std 1003.2d-1994:

```
_POSIX2_PBS
_POSIX2_PBS_ACCOUNTING
_POSIX2_PBS_LOCATE
_POSIX2_PBS_MESSAGE
_POSIX2_PBS_TRACK
```

46696 The following *sysconf()* variables and their associated names are added for alignment with
 46697 IEEE Std 1003.1q-2000:

```
_POSIX_TRACE
_POSIX_TRACE_EVENT_FILTER
_POSIX_TRACE_INHERIT
_POSIX_TRACE_LOG
```

46702 The macros associated with the *c89* programming models are marked LEGACY, and new
 46703 equivalent macros associated with *c99* are introduced.

46704 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/62 is applied, updating the 1
 46705 DESCRIPTION to denote that the _PC* and _SC* symbols are now required to be supported. A 1
 46706 corresponding change has been made in the Base Definitions volume of IEEE Std 1003.1-2001. 1
 46707 The deletion in the second paragraph removes some duplicated text. Additional symbols that 1
 46708 were erroneously omitted from this reference page have been added. 1

46709 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/63 is applied, making it clear in the 1
 46710 RETURN VALUE section that the value returned for *sysconf(_SC_OPEN_MAX)* may change if a 1
 46711 call to *setrlimit()* adjusts the RLIMIT_NOFILE soft limit. 1

46712 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/134 is applied, updating the 2
 46713 DESCRIPTION to remove an erroneous entry for _POSIX_SYMLOOP_MAX. This corrects an 2

46714	error in IEEE Std 1003.1-2001/Cor 1-2002.	2
46715	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/135 is applied, removing _POSIX_FILE_LOCKING, _POSIX_MULTI_PROCESS, _POSIX2_C_VERSION, and _XOPEN_XCU_VERSION (and their associated _SC_* variables) from the DESCRIPTION and APPLICATION USAGE sections.	2
46719	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/136 is applied, adding _POSIX_SS_REPL_MAX, _POSIX_TRACE_EVENT_NAME_MAX, _POSIX_TRACE_NAME_MAX, _POSIX_TRACE_SYS_MAX, and _POSIX_TRACE_USER_EVENT_MAX (and their associated _SC_* variables) to the DESCRIPTION. The RETURN VALUE and APPLICATION USAGE sections are updated to note that if variables are dependent on unsupported options, the results are unspecified.	2
46725	IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/137 is applied, removing _REGEX_VERSION and _SC_REGEX_VERSION.	2
46726		2

46727 NAME

46728 **syslog** — log a message

46729 SYNOPSIS

46730 XSI `#include <syslog.h>`

46731 `void syslog(int priority, const char *message, ... /* argument */);`

46732

46733 DESCRIPTION

46734 Refer to *closelog()*.

46735 NAME

46736 system — issue a command

46737 SYNOPSIS

46738

```
#include <stdlib.h>
```


46739

```
int system(const char *command);
```

46740 DESCRIPTION

46741 CX The functionality described on this reference page is aligned with the ISO C standard. Any
46742 conflict between the requirements described here and the ISO C standard is unintentional. This
46743 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.46744 If *command* is a null pointer, the *system()* function shall determine whether the host environment
46745 has a command processor. If *command* is not a null pointer, the *system()* function shall pass the
46746 string pointed to by *command* to that command processor to be executed in an implementation-
46747 defined manner; this might then cause the program calling *system()* to behave in a non-
46748 conforming manner or to terminate.46749 CX The environment of the executed command shall be as if a child process were created using
46750 *fork()*, and the child process invoked the *sh* utility using *execl()* as follows:

```
execl(<shell path>, "sh", "-c", command, (char *)0);
```

46752 where *<shell path>* is an unspecified pathname for the *sh* utility.46753 The *system()* function shall ignore the SIGINT and SIGQUIT signals, and shall block the
46754 SIGCHLD signal, while waiting for the command to terminate. If this might cause the
46755 application to miss a signal that would have killed it, then the application should examine the
46756 return value from *system()* and take whatever action is appropriate to the application if the
46757 command terminated due to receipt of a signal.46758 The *system()* function shall not affect the termination status of any child of the calling processes
46759 other than the process or processes it itself creates.46760 The *system()* function shall not return until the child process has terminated.

46761 RETURN VALUE

46762 If *command* is a null pointer, *system()* shall return non-zero to indicate that a command processor
46763 CX is available, or zero if none is available. The *system()* function shall always return non-zero when
46764 *command* is NULL.46765 CX If *command* is not a null pointer, *system()* shall return the termination status of the command
46766 language interpreter in the format specified by *waitpid()*. The termination status shall be as
46767 defined for the *sh* utility; otherwise, the termination status is unspecified. If some error prevents
46768 the command language interpreter from executing after the child process is created, the return
46769 value from *system()* shall be as if the command language interpreter had terminated using
46770 *exit(127)* or *_exit(127)*. If a child process cannot be created, or if the termination status for the
46771 command language interpreter cannot be obtained, *system()* shall return -1 and set *errno* to
46772 indicate the error.

46773 ERRORS

46774 CX The *system()* function may set *errno* values as described by *fork()*.46775 In addition, *system()* may fail if:46776 CX [ECHILD] The status of the child process created by *system()* is no longer available.

46777 EXAMPLES

46778 None.

46779 APPLICATION USAGE

46780 If the return value of *system()* is not -1 , its value can be decoded through the use of the macros described in `<sys/wait.h>`. For convenience, these macros are also provided in `<stdlib.h>`.

46782 Note that, while *system()* must ignore SIGINT and SIGQUIT and block SIGCHLD while waiting for the child to terminate, the handling of signals in the executed command is as specified by *fork()* and *exec*. For example, if SIGINT is being caught or is set to SIG_DFL when *system()* is called, then the child is started with SIGINT handling set to SIG_DFL.

46786 Ignoring SIGINT and SIGQUIT in the parent process prevents coordination problems (two processes reading from the same terminal, for example) when the executed command ignores or catches one of the signals. It is also usually the correct action when the user has given a command to the application to be executed synchronously (as in the '`!`` command in many interactive applications). In either case, the signal should be delivered only to the child process, not to the application itself. There is one situation where ignoring the signals might have less than the desired effect. This is when the application uses *system()* to perform some task invisible to the user. If the user typed the interrupt character ("`^C`", for example) while *system()* is being used in this way, one would expect the application to be killed, but only the executed command is killed. Applications that use *system()* in this way should carefully check the return status from *system()* to see if the executed command was successful, and should take appropriate action when the command fails.

46798 Blocking SIGCHLD while waiting for the child to terminate prevents the application from catching the signal and obtaining status from *system()*'s child process before *system()* can get the status itself.

46801 The context in which the utility is ultimately executed may differ from that in which *system()* was called. For example, file descriptors that have the FD_CLOEXEC flag set are closed, and the process ID and parent process ID are different. Also, if the executed utility changes its environment variables or its current working directory, that change is not reflected in the caller's context.

46806 There is no defined way for an application to find the specific path for the shell. However, *confstr()* can provide a value for *PATH* that is guaranteed to find the *sh* utility.

46808 RATIONALE

46809 The *system()* function should not be used by programs that have set user (or group) ID privileges. The *fork()* and *exec* family of functions (except *execvp()* and *execv()*), should be used instead. This prevents any unforeseen manipulation of the environment of the user that could cause execution of commands not anticipated by the calling program.

46813 There are three levels of specification for the *system()* function. The ISO C standard gives the most basic. It requires that the function exists, and defines a way for an application to query whether a command language interpreter exists. It says nothing about the command language or the environment in which the command is interpreted.

46817 IEEE Std 1003.1-2001 places additional restrictions on *system()*. It requires that if there is a command language interpreter, the environment must be as specified by *fork()* and *exec*. This ensures, for example, that close-on-exec works, that file locks are not inherited, and that the process ID is different. It also specifies the return value from *system()* when the command line can be run, thus giving the application some information about the command's completion status.

46823 Finally, IEEE Std 1003.1-2001 requires the command to be interpreted as in the shell command
46824 language defined in the Shell and Utilities volume of IEEE Std 1003.1-2001.

46825 Note that, *system(NULL)* is required to return non-zero, indicating that there is a command
46826 language interpreter. At first glance, this would seem to conflict with the ISO C standard which
46827 allows *system(NULL)* to return zero. There is no conflict, however. A system must have a
46828 command language interpreter, and is non-conforming if none is present. It is therefore
46829 permissible for the *system()* function on such a system to implement the behavior specified by
46830 the ISO C standard as long as it is understood that the implementation does not conform to
46831 IEEE Std 1003.1-2001 if *system(NULL)* returns zero.

46832 It was explicitly decided that when *command* is NULL, *system()* should not be required to check
46833 to make sure that the command language interpreter actually exists with the correct mode, that
46834 there are enough processes to execute it, and so on. The call *system(NULL)* could, theoretically,
46835 check for such problems as too many existing child processes, and return zero. However, it
46836 would be inappropriate to return zero due to such a (presumably) transient condition. If some
46837 condition exists that is not under the control of this application and that would cause any
46838 *system()* call to fail, that system has been rendered non-conforming.

46839 Early drafts required, or allowed, *system()* to return with *errno* set to [EINTR] if it was
46840 interrupted with a signal. This error return was removed, and a requirement that *system()* not
46841 return until the child has terminated was added. This means that if a *waitpid()* call in *system()*
46842 exits with *errno* set to [EINTR], *system()* must reissue the *waitpid()*. This change was made for
46843 two reasons:

- 46844 1. There is no way for an application to clean up if *system()* returns [EINTR], short of calling
46845 *wait()*, and that could have the undesirable effect of returning the status of children other
46846 than the one started by *system()*.
- 46847 2. While it might require a change in some historical implementations, those
46848 implementations already have to be changed because they use *wait()* instead of *waitpid()*.

46849 Note that if the application is catching SIGCHLD signals, it will receive such a signal before a
46850 successful *system()* call returns.

46851 To conform to IEEE Std 1003.1-2001, *system()* must use *waitpid()*, or some similar function,
46852 instead of *wait()*.

46853 The following code sample illustrates how *system()* might be implemented on an
46854 implementation conforming to IEEE Std 1003.1-2001.

```
46855 #include <signal.h>
46856 int system(const char *cmd)
46857 {
46858     int stat;
46859     pid_t pid;
46860     struct sigaction sa, savintr, savequit;
46861     sigset_t saveblock;
46862     if (cmd == NULL)
46863         return(1);
46864     sa.sa_handler = SIG_IGN;
46865     sigemptyset(&sa.sa_mask);
46866     sa.sa_flags = 0;
46867     sigemptyset(&savintr.sa_mask);
46868     sigemptyset(&savequit.sa_mask);
46869     sigaction(SIGINT, &sa, &savintr);
46870     sigaction(SIGQUIT, &sa, &savequit);
```

```

46871     sigaddset(&sa.sa_mask, SIGCHLD);
46872     sigprocmask(SIG_BLOCK, &sa.sa_mask, &saveblock);
46873     if ((pid = fork()) == 0) {
46874         sigaction(SIGINT, &savintr, (struct sigaction *)0);
46875         sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
46876         sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
46877         execl("/bin/sh", "sh", "-c", cmd, (char *)0);
46878         _exit(127);
46879     }
46880     if (pid == -1) {
46881         stat = -1; /* errno comes from fork() */
46882     } else {
46883         while (waitpid(pid, &stat, 0) == -1) {
46884             if (errno != EINTR){
46885                 stat = -1;
46886                 break;
46887             }
46888         }
46889     }
46890     sigaction(SIGINT, &savintr, (struct sigaction *)0);
46891     sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
46892     sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
46893     return(stat);
46894 }

```

Note that, while a particular implementation of *system()* (such as the one above) can assume a particular path for the shell, such a path is not necessarily valid on another system. The above example is not portable, and is not intended to be.

One reviewer suggested that an implementation of *system()* might want to use an environment variable such as *SHELL* to determine which command interpreter to use. The supposed implementation would use the default command interpreter if the one specified by the environment variable was not available. This would allow a user, when using an application that prompts for command lines to be processed using *system()*, to specify a different command interpreter. Such an implementation is discouraged. If the alternate command interpreter did not follow the command line syntax specified in the Shell and Utilities volume of IEEE Std 1003.1-2001, then changing *SHELL* would render *system()* non-conforming. This would affect applications that expected the specified behavior from *system()*, and since the Shell and Utilities volume of IEEE Std 1003.1-2001 does not mention that *SHELL* affects *system()*, the application would not know that it needed to unset *SHELL*.

46909 FUTURE DIRECTIONS

46910 None.

46911 SEE ALSO

46912 *exec*, *pipe()*, *waitpid()*, the Base Definitions volume of IEEE Std 1003.1-2001, *<limits.h>*,
 46913 *<signal.h>*, *<stdlib.h>*, *<sys/wait.h>*, the Shell and Utilities volume of IEEE Std 1003.1-2001, *sh*

46914 CHANGE HISTORY

46915 First released in Issue 1. Derived from Issue 1 of the SVID.

46916 Issue 6

46917 Extensions beyond the ISO C standard are marked.

46918 NAME

46919 tan, tanf, tanl — tangent function

46920 SYNOPSIS

```
46921     #include <math.h>
46922
46923     double tan(double x);
46924     float tanf(float x);
46925     long double tanl(long double x);
```

46925 DESCRIPTION

46926 CX 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.

46929 These functions shall compute the tangent of their argument *x*, measured in radians.

46930 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.

46934 RETURN VALUE

46935 Upon successful completion, these functions shall return the tangent of *x*.

46936 If the correct value would cause underflow, and is not representable, a range error may occur, and either 0.0 (if supported), or an implementation-defined value shall be returned.

46938 MX If *x* is NaN, a NaN shall be returned.

46939 If *x* is ±0, *x* shall be returned.

46940 If *x* is subnormal, a range error may occur and *x* should be returned.

46941 If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

46943 If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

46945 XSI If the correct value would cause overflow, a range error shall occur and *tan()*, *tanf()*, and *tanl()* shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL, respectively, with the same sign as the correct value of the function. 1
46946 1
46947 1

46948 ERRORS

46949 These functions shall fail if:

46950 MX Domain Error The value of *x* is ±Inf.

46951 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.

46955 XSI Range Error The result overflows

46956 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.

46960 These functions may fail if:

46961 MX Range Error The result underflows, or the value of x is subnormal.

46962
46963
46964
46965 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 underflow floating-point exception shall be raised.

46966 EXAMPLES

46967 Taking the Tangent of a 45-Degree Angle

```
46968 #include <math.h>
46969 ...
46970 double radians = 45.0 * M_PI / 180;
46971 double result;
46972 ...
46973 result = tan (radians);
```

46974 APPLICATION USAGE

46975 There are no known floating-point representations such that for a normal argument, $\tan(x)$ is either overflow or underflow.

46977 These functions may lose accuracy when their argument is near a multiple of $\pi/2$ or is far from 0.0.

46979 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling & MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

46981 RATIONALE

46982 None.

46983 FUTURE DIRECTIONS

46984 None.

46985 SEE ALSO

46986 *atan()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

46988 CHANGE HISTORY

46989 First released in Issue 1. Derived from Issue 1 of the SVID.

46990 Issue 5

46991 The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

46993 Issue 6

46994 The *tanf()* and *tanl()* functions are added for alignment with the ISO/IEC 9899: 1999 standard.

46995 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899: 1999 standard.

46997 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are marked.

46999 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/64 is applied, correcting the last 1 paragraph in the RETURN VALUE section. 1

47001 NAME

47002 tanh, tanhf, tanhl — hyperbolic tangent functions

47003 SYNOPSIS

```
47004     #include <math.h>
47005
47006     double tanh(double x);
47007     float tanhf(float x);
47008     long double tanhl(long double x);
```

47008 DESCRIPTION

47009 CX 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.

47012 These functions shall compute the hyperbolic tangent of their argument *x*.

47013 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.

47017 RETURN VALUE

47018 Upon successful completion, these functions shall return the hyperbolic tangent of *x*.

47019 MX If *x* is NaN, a NaN shall be returned.

47020 If *x* is ±0, *x* shall be returned.

47021 If *x* is ±Inf, ±1 shall be returned.

47022 If *x* is subnormal, a range error may occur and *x* should be returned.

47023 ERRORS

47024 These functions may fail if:

47025 MX Range Error The value of *x* is subnormal.

47026 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 underflow floating-point exception shall be raised.

47030 EXAMPLES

47031 None.

47032 APPLICATION USAGE

47033 On error, the expressions (*math_errhandling* & MATH_ERRNO) and (*math_errhandling* & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

47035 RATIONALE

47036 None.

47037 FUTURE DIRECTIONS

47038 None.

47039 SEE ALSO

47040 *atanh()*, *feclearexcept()*, *fetestexcept()*, *isnan()*, *tan()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, *<math.h>*

47043 CHANGE HISTORY

47044 First released in Issue 1. Derived from Issue 1 of the SVID.

47045 Issue 5

47046 The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

47048 Issue 6

47049 The *tanhf()* and *tanhl()* functions are added for alignment with the ISO/IEC 9899: 1999 standard.

47050 The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899: 1999 standard.

47052 IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are marked.

47053

47054 NAME

47055 **tanl** — tangent function

47056 SYNOPSIS

47057 #include <math.h>

47058 long double tanl(long double x);

47059 DESCRIPTION

47060 Refer to *tan()*.

47061 NAME

47062 *tcdrain* — wait for transmission of output

47063 SYNOPSIS

```
47064        #include <termios.h>
47065        int tcdrain(int fildes);
```

47066 DESCRIPTION

47067 The *tcdrain*() function shall block until all output written to the object referred to by *fildes* is transmitted. The *fildes* argument is an open file descriptor associated with a terminal.

47069 Any attempts to use *tcdrain*() from a process which is a member of a background process group on a *fildes* 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.

47073 RETURN VALUE

47074 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

47076 ERRORS

47077 The *tcdrain*() function shall fail if:

47078 [EBADF] The *fildes* argument is not a valid file descriptor.

47079 [EINTR] A signal interrupted *tcdrain*().

47080 [ENOTTY] The file associated with *fildes* is not a terminal.

47081 The *tcdrain*() function may fail if:

47082 [EIO] The process group of the writing process is orphaned, and the writing process
47083 is not ignoring or blocking SIGTTOU.

47084 EXAMPLES

47085 None.

47086 APPLICATION USAGE

47087 None.

47088 RATIONALE

47089 None.

47090 FUTURE DIRECTIONS

47091 None.

47092 SEE ALSO

47093 *tflush*(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal
47094 Interface, <termios.h>, <unistd.h>

47095 CHANGE HISTORY

47096 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

47097 Issue 6

47098 The following new requirements on POSIX implementations derive from alignment with the
47099 Single UNIX Specification:

- 47100 • In the DESCRIPTION, the final paragraph is no longer conditional on
47101 _POSIX_JOB_CONTROL. This is a FIPS requirement.
- 47102 • The [EIO] error is added.

47103 NAME

47104 tcflow — suspend or restart the transmission or reception of data

47105 SYNOPSIS

```
47106        #include <termios.h>
47107        int tcflow(int fildes, int action);
```

47108 DESCRIPTION

47109 The *tcflow()* function shall suspend or restart transmission or reception of data on the object
47110 referred to by *fildes*, depending on the value of *action*. The *fildes* argument is an open file
47111 descriptor associated with a terminal.

- 47112 • If *action* is TCOOFF, output shall be suspended.
- 47113 • If *action* is TCOON, suspended output shall be restarted.
- 47114 • If *action* is TCIOFF, the system shall transmit a STOP character, which is intended to cause
47115 the terminal device to stop transmitting data to the system.
- 47116 • If *action* is TCION, the system shall transmit a START character, which is intended to cause
47117 the terminal device to start transmitting data to the system.

47118 The default on the opening of a terminal file is that neither its input nor its output are
47119 suspended.

47120 Attempts to use *tcflow()* from a process which is a member of a background process group on a
47121 *fildes* associated with its controlling terminal, shall cause the process group to be sent a
47122 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process
47123 shall be allowed to perform the operation, and no signal is sent.

47124 RETURN VALUE

47125 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
47126 indicate the error.

47127 ERRORS

47128 The *tcflow()* function shall fail if:

47129 [EBADF] The *fildes* argument is not a valid file descriptor.
47130 [EINVAL] The *action* argument is not a supported value.
47131 [ENOTTY] The file associated with *fildes* is not a terminal.

47132 The *tcflow()* function may fail if:

47133 [EIO] The process group of the writing process is orphaned, and the writing process
47134 is not ignoring or blocking SIGTTOU.

47135 EXAMPLES

47136 None.

47137 APPLICATION USAGE

47138 None.

47139 RATIONALE

47140 None.

47141 FUTURE DIRECTIONS

47142 None.

47143 SEE ALSO

47144 *tcsendbreak()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal
47145 Interface, <termios.h>, <unistd.h>

47146 CHANGE HISTORY

47147 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

47148 Issue 6

47149 The following new requirements on POSIX implementations derive from alignment with the
47150 Single UNIX Specification:

- 47151 • The [EIO] error is added.

47152 NAME

47153 *tcflush* — flush non-transmitted output data, non-read input data, or both

47154 SYNOPSIS

```
47155        #include <termios.h>
47156        int tcflush(int fildes, int queue_selector);
```

47157 DESCRIPTION

47158 Upon successful completion, *tcflush()* shall discard data written to the object referred to by *fildes* (an open file descriptor associated with a terminal) but not transmitted, or data received but not read, depending on the value of *queue_selector*:

- 47161 • If *queue_selector* is TCIFLUSH, it shall flush data received but not read.
- 47162 • If *queue_selector* is TCOFLUSH, it shall flush data written but not transmitted.
- 47163 • If *queue_selector* is TCIOFLUSH, it shall flush both data received but not read and data written but not transmitted.

47165 Attempts to use *tcflush()* from a process which is a member of a background process group on a *fildes* 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.

47169 RETURN VALUE

47170 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

47172 ERRORS

47173 The *tcflush()* function shall fail if:

47174 [EBADF] The *fildes* argument is not a valid file descriptor.
47175 [EINVAL] The *queue_selector* argument is not a supported value.
47176 [ENOTTY] The file associated with *fildes* is not a terminal.

47177 The *tcflush()* function may fail if:

47178 [EIO] The process group of the writing process is orphaned, and the writing process
47179 is not ignoring or blocking SIGTTOU.

47180 EXAMPLES

47181 None.

47182 APPLICATION USAGE

47183 None.

47184 RATIONALE

47185 None.

47186 FUTURE DIRECTIONS

47187 None.

47188 SEE ALSO

47189 *tcdrain()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal
47190 Interface, <termios.h>, <unistd.h>

47191 CHANGE HISTORY

47192 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

47193 Issue 6

47194 The Open Group Corrigendum U035/1 is applied. In the ERRORS and APPLICATION USAGE sections, references to *tcflow()* are replaced with *tcflush()*.

47196 The following new requirements on POSIX implementations derive from alignment with the
47197 Single UNIX Specification:

- 47198 • In the DESCRIPTION, the final paragraph is no longer conditional on
47199 _POSIX_JOB_CONTROL. This is a FIPS requirement.
- 47200 • The [EIO] error is added.

47201 NAME

47202 tcgetattr — get the parameters associated with the terminal

47203 SYNOPSIS

```
47204        #include <termios.h>
47205        int tcgetattr(int fd, struct termios *termios_p);
```

47206 DESCRIPTION

47207 The *tcgetattr()* function shall get the parameters associated with the terminal referred to by *fd* and store them in the **termios** structure referenced by *termios_p*. The *fd* argument is an open file descriptor associated with a terminal.

47210 The *termios_p* argument is a pointer to a **termios** structure.

47211 The *tcgetattr()* operation is allowed from any process.

47212 If the terminal device supports different input and output baud rates, the baud rates stored in the **termios** structure returned by *tcgetattr()* shall reflect the actual baud rates, even if they are equal. If differing baud rates are not supported, the rate returned as the output baud rate shall be the actual baud rate. If the terminal device does not support split baud rates, the input baud rate stored in the **termios** structure shall be the output rate (as one of the symbolic values).

47217 RETURN VALUE

47218 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

47220 ERRORS

47221 The *tcgetattr()* function shall fail if:

47222 [EBADF] The *fd* argument is not a valid file descriptor.

47223 [ENOTTY] The file associated with *fd* is not a terminal.

47224 EXAMPLES

47225 None.

47226 APPLICATION USAGE

47227 None.

47228 RATIONALE

47229 Care must be taken when changing the terminal attributes. Applications should always do a *tcgetattr()*, save the **termios** structure values returned, and then do a *tcsetattr()*, changing only the necessary fields. The application should use the values saved from the *tcgetattr()* to reset the terminal state whenever it is done with the terminal. This is necessary because terminal attributes apply to the underlying port and not to each individual open instance; that is, all processes that have used the terminal see the latest attribute changes.

47235 A program that uses these functions should be written to catch all signals and take other appropriate actions to ensure that when the program terminates, whether planned or not, the terminal device's state is restored to its original state.

47238 Existing practice dealing with error returns when only part of a request can be honored is based on calls to the *ioctl()* function. In historical BSD and System V implementations, the corresponding *ioctl()* returns zero if the requested actions were semantically correct, even if some of the requested changes could not be made. Many existing applications assume this behavior and would no longer work correctly if the return value were changed from zero to -1 in this case.

47244 Note that either specification has a problem. When zero is returned, it implies everything
47245 succeeded even if some of the changes were not made. When -1 is returned, it implies
47246 everything failed even though some of the changes were made.

47247 Applications that need all of the requested changes made to work properly should follow
47248 *tcsetattr()* with a call to *tcgetattr()* and compare the appropriate field values.

47249 FUTURE DIRECTIONS

47250 None.

47251 SEE ALSO

47252 *tcsetattr()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal
47253 Interface, <termios.h>

47254 CHANGE HISTORY

47255 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

47256 Issue 6

47257 In the DESCRIPTION, the rate returned as the input baud rate shall be the output rate.
47258 Previously, the number zero was also allowed but was obsolescent.

47259 NAME

47260 tcgetpgrp — get the foreground process group ID

47261 SYNOPSIS

```
47262        #include <unistd.h>
47263        pid_t tcgetpgrp(int fildes);
```

47264 DESCRIPTION

47265 The *tcgetpgrp()* function shall return the value of the process group ID of the foreground process group associated with the terminal.

47267 If there is no foreground process group, *tcgetpgrp()* shall return a value greater than 1 that does not match the process group ID of any existing process group.

47269 The *tcgetpgrp()* function is allowed from a process that is a member of a background process group; however, the information may be subsequently changed by a process that is a member of a foreground process group.

47272 RETURN VALUE

47273 Upon successful completion, *tcgetpgrp()* shall return the value of the process group ID of the foreground process associated with the terminal. Otherwise, -1 shall be returned and *errno* set to indicate the error.

47276 ERRORS

47277 The *tcgetpgrp()* function shall fail if:

47278 [EBADF] The *fildes* argument is not a valid file descriptor.

47279 [ENOTTY] The calling process does not have a controlling terminal, or the file is not the controlling terminal.

47281 EXAMPLES

47282 None.

47283 APPLICATION USAGE

47284 None.

47285 RATIONALE

47286 None.

47287 FUTURE DIRECTIONS

47288 None.

47289 SEE ALSO

47290 *setsid()*, *setpgid()*, *tcsetpgrp()*, the Base Definitions volume of IEEE Std 1003.1-2001,
47291 <*sys/types.h*>, <*unistd.h*>

47292 CHANGE HISTORY

47293 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

47294 Issue 6

47295 In the SYNOPSIS, the optional include of the <*sys/types.h*> header is removed.

47296 The following new requirements on POSIX implementations derive from alignment with the
47297 Single UNIX Specification:

- 47298 • The requirement to include <*sys/types.h*> has been removed. Although <*sys/types.h*> was
47299 required for conforming implementations of previous POSIX specifications, it was not
47300 required for UNIX applications.

47301
47302

- In the DESCRIPTION, text previously conditional on support for _POSIX_JOB_CONTROL is now mandatory. This is a FIPS requirement.

47303 NAME

47304 tcgetsid — get the process group ID for the session leader for the controlling terminal

47305 SYNOPSIS

47306 XSI #include <termios.h>

47307 pid_t tcgetsid(int *fildes*);

47308

47309 DESCRIPTION

47310 The *tcgetsid()* function shall obtain the process group ID of the session for which the terminal
47311 specified by *fildes* is the controlling terminal.

47312 RETURN VALUE

47313 Upon successful completion, *tcgetsid()* shall return the process group ID associated with the
47314 terminal. Otherwise, a value of (**pid_t**)–1 shall be returned and *errno* set to indicate the error.

47315 ERRORS

47316 The *tcgetsid()* function shall fail if:

47317 [EBADF] The *fildes* argument is not a valid file descriptor.

47318 [ENOTTY] The calling process does not have a controlling terminal, or the file is not the
47319 controlling terminal.

47320 EXAMPLES

47321 None.

47322 APPLICATION USAGE

47323 None.

47324 RATIONALE

47325 None.

47326 FUTURE DIRECTIONS

47327 None.

47328 SEE ALSO

47329 The Base Definitions volume of IEEE Std 1003.1-2001, <**termios.h**>

47330 CHANGE HISTORY

47331 First released in Issue 4, Version 2.

47332 Issue 5

47333 Moved from X/OPEN UNIX extension to BASE.

47334 The [EACCES] error has been removed from the list of mandatory errors, and the description of
47335 [ENOTTY] has been reworded.

47336 NAME

47337 *tcsendbreak* — send a break for a specific duration

47338 SYNOPSIS

```
47339        #include <termios.h>
47340        int tcsendbreak(int fildes, int duration);
```

47341 DESCRIPTION

47342 If the terminal is using asynchronous serial data transmission, *tcsendbreak()* shall cause
47343 transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is 0, it
47344 shall cause transmission of zero-valued bits for at least 0.25 seconds, and not more than 0.5
47345 seconds. If *duration* is not 0, it shall send zero-valued bits for an implementation-defined period
47346 of time.

47347 The *fildes* argument is an open file descriptor associated with a terminal.

47348 If the terminal is not using asynchronous serial data transmission, it is implementation-defined
47349 whether *tcsendbreak()* sends data to generate a break condition or returns without taking any
47350 action.

47351 Attempts to use *tcsendbreak()* from a process which is a member of a background process group
47352 on a *fildes* associated with its controlling terminal shall cause the process group to be sent a
47353 SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process
47354 shall be allowed to perform the operation, and no signal is sent.

47355 RETURN VALUE

47356 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
47357 indicate the error.

47358 ERRORS

47359 The *tcsendbreak()* function shall fail if:

47360 [EBADF] The *fildes* argument is not a valid file descriptor.

47361 [ENOTTY] The file associated with *fildes* is not a terminal.

47362 The *tcsendbreak()* function may fail if:

47363 [EIO] The process group of the writing process is orphaned, and the writing process
47364 is not ignoring or blocking SIGTTOU.

47365 EXAMPLES

47366 None.

47367 APPLICATION USAGE

47368 None.

47369 RATIONALE

47370 None.

47371 FUTURE DIRECTIONS

47372 None.

47373 SEE ALSO

47374 The Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface,
47375 <termios.h>, <unistd.h>

47376 CHANGE HISTORY

47377 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

47378 Issue 6

47379 The following new requirements on POSIX implementations derive from alignment with the
47380 Single UNIX Specification:

- 47381 • In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now
47382 mandated. This is a FIPS requirement.
47383 • The [EIO] error is added.

47384 NAME

47385 tcsetattr — set the parameters associated with the terminal

47386 SYNOPSIS

```
47387 #include <termios.h>
47388 int tcsetattr(int fildes, int optional_actions,
47389     const struct termios *termios_p);
```

47390 DESCRIPTION

47391 The *tcsetattr()* function shall set the parameters associated with the terminal referred to by the
47392 open file descriptor *fildes* (an open file descriptor associated with a terminal) from the **termios**
47393 structure referenced by *termios_p* as follows:

- 47394 • If *optional_actions* is TCSANOW, the change shall occur immediately.
- 47395 • If *optional_actions* is TCSADRAIN, the change shall occur after all output written to *fildes* is
47396 transmitted. This function should be used when changing parameters that affect output.
- 47397 • If *optional_actions* is TCSAFLUSH, the change shall occur after all output written to *fildes* is
47398 transmitted, and all input so far received but not read shall be discarded before the change is
47399 made.

47400 If the output baud rate stored in the **termios** structure pointed to by *termios_p* is the zero baud
47401 rate, B0, the modem control lines shall no longer be asserted. Normally, this shall disconnect the
47402 line.

47403 If the input baud rate stored in the **termios** structure pointed to by *termios_p* is 0, the input baud
47404 rate given to the hardware is the same as the output baud rate stored in the **termios** structure.

47405 The *tcsetattr()* function shall return successfully if it was able to perform any of the requested
47406 actions, even if some of the requested actions could not be performed. It shall set all the
47407 attributes that the implementation supports as requested and leave all the attributes not
47408 supported by the implementation unchanged. If no part of the request can be honored, it shall
47409 return -1 and set *errno* to [EINVAL]. If the input and output baud rates differ and are a
47410 combination that is not supported, neither baud rate shall be changed. A subsequent call to
47411 *tcgetattr()* shall return the actual state of the terminal device (reflecting both the changes made
47412 and not made in the previous *tcsetattr()* call). The *tcsetattr()* function shall not change the values
47413 found in the **termios** structure under any circumstances.

47414 The effect of *tcsetattr()* is undefined if the value of the **termios** structure pointed to by *termios_p*
47415 was not derived from the result of a call to *tcgetattr()* on *fildes*; an application should modify
47416 only fields and flags defined by this volume of IEEE Std 1003.1-2001 between the call to
47417 *tcgetattr()* and *tcsetattr()*, leaving all other fields and flags unmodified.

47418 No actions defined by this volume of IEEE Std 1003.1-2001, other than a call to *tcsetattr()* or a
47419 close of the last file descriptor in the system associated with this terminal device, shall cause any
47420 of the terminal attributes defined by this volume of IEEE Std 1003.1-2001 to change.

47421 If *tcsetattr()* is called from a process which is a member of a background process group on a
47422 *fildes* associated with its controlling terminal:

- 47423 • If the calling process is blocking or ignoring SIGTTOU signals, the operation completes
47424 normally and no signal is sent.
- 47425 • Otherwise, a SIGTTOU signal shall be sent to the process group.

47426 RETURN VALUE

47427 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
47428 indicate the error.

47429 ERRORS

47430 The *tcsetattr()* function shall fail if:

47431 [EBADF] The *fd* argument is not a valid file descriptor.
47432 [EINTR] A signal interrupted *tcsetattr()*.
47433 [EINVAL] The *optional_actions* argument is not a supported value, or an attempt was
47434 made to change an attribute represented in the **termios** structure to an
47435 unsupported value.

47436 [ENOTTY] The file associated with *fd* is not a terminal.

47437 The *tcsetattr()* function may fail if:

47438 [EIO] The process group of the writing process is orphaned, and the writing process
47439 is not ignoring or blocking SIGTTOU.

47440 EXAMPLES

47441 None.

47442 APPLICATION USAGE

47443 If trying to change baud rates, applications should call *tcsetattr()* then call *tcgetattr()* in order to
47444 determine what baud rates were actually selected.

47445 RATIONALE

47446 The *tcsetattr()* function can be interrupted in the following situations:

- 47447 • It is interrupted while waiting for output to drain.
- 47448 • It is called from a process in a background process group and SIGTTOU is caught.

47449 See also the RATIONALE section in *tcgetattr()*.

47450 FUTURE DIRECTIONS

47451 Using an input baud rate of 0 to set the input rate equal to the output rate may not necessarily be
47452 supported in a future version of this volume of IEEE Std 1003.1-2001.

47453 SEE ALSO

47454 *cgetispeed()*, *tcgetattr()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11,
47455 General Terminal Interface, <**termios.h**>, <**unistd.h**>

47456 CHANGE HISTORY

47457 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

47458 Issue 6

47459 The following new requirements on POSIX implementations derive from alignment with the
47460 Single UNIX Specification:

- 47461 • In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now
47462 mandated. This is a FIPS requirement.
- 47463 • The [EIO] error is added.

47464
47465

In the DESCRIPTION, the text describing use of *tcsetattr()* from a process which is a member of a background process group is clarified.

47466 NAME

47467 *tcsetpgrp* — set the foreground process group ID

47468 SYNOPSIS

```
47469        #include <unistd.h>
47470        int tcsetpgrp(int fildes, pid_t pgid_id);
```

47471 DESCRIPTION

47472 If the process has a controlling terminal, *tcsetpgrp()* shall set the foreground process group ID associated with the terminal to *pgid_id*. The application shall ensure that the file associated with *fildes* is the controlling terminal of the calling process and the controlling terminal is currently associated with the session of the calling process. The application shall ensure that the value of *pgid_id* matches a process group ID of a process in the same session as the calling process.

47477 Attempts to use *tcsetpgrp()* from a process which is a member of a background process group on a *fildes* 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.

47481 RETURN VALUE

47482 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

47484 ERRORS

47485 The *tcsetpgrp()* function shall fail if:

47486 [EBADF]	The <i>fildes</i> argument is not a valid file descriptor.
47487 [EINVAL]	This implementation does not support the value in the <i>pgid_id</i> argument.
47488 [ENOTTY]	The calling process does not have a controlling terminal, or the file is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.
47491 [EPERM]	The value of <i>pgid_id</i> is a value supported by the implementation, but does not match the process group ID of a process in the same session as the calling process.

47494 EXAMPLES

47495 None.

47496 APPLICATION USAGE

47497 None.

47498 RATIONALE

47499 None.

47500 FUTURE DIRECTIONS

47501 None.

47502 SEE ALSO

47503 *tcgetpgrp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <*sys/types.h*>, <*unistd.h*>

47504 CHANGE HISTORY

47505 First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

47506 **Issue 6**

- 47507 In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.
- 47508 The following new requirements on POSIX implementations derive from alignment with the
47509 Single UNIX Specification:
- 47510 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
47511 required for conforming implementations of previous POSIX specifications, it was not
47512 required for UNIX applications.
 - 47513 • In the DESCRIPTION and ERRORS sections, text previously conditional on
47514 _POSIX_JOB_CONTROL is now mandated. This is a FIPS requirement.
- 47515 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
- 47516 The Open Group Corrigendum U047/4 is applied.

47517 NAME

47518 *tdelete, tfind, tsearch, twalk* — manage a binary search tree

47519 SYNOPSIS

```
47520 XSI #include <search.h>
47521
47522     void *tdelete(const void *restrict key, void **restrict rootp,
47523         int(*compar)(const void *, const void *));
47524     void *tfind(const void *key, void *const *rootp,
47525         int(*compar)(const void *, const void *));
47526     void *tsearch(const void *key, void **rootp,
47527         int (*compar)(const void *, const void *));
47528     void twalk(const void *root,
47529         void (*action)(const void *, VISIT, int));
```

47530 DESCRIPTION

47531 The *tdelete()*, *tfind()*, *tsearch()*, and *twalk()* functions manipulate binary search trees.
 47532 Comparisons are made with a user-supplied routine, the address of which is passed as the
 47533 *compar* argument. This routine is called with two arguments, which are the pointers to the
 47534 elements being compared. The application shall ensure that the user-supplied routine returns an
 47535 integer less than, equal to, or greater than 0, according to whether the first argument is to be
 47536 considered less than, equal to, or greater than the second argument. The comparison function
 47537 need not compare every byte, so arbitrary data may be contained in the elements in addition to
 47538 the values being compared.

47539 The *tsearch()* function shall build and access the tree. The *key* argument is a pointer to an element
 47540 to be accessed or stored. If there is a node in the tree whose element is equal to the value pointed
 47541 to by *key*, a pointer to this found node shall be returned. Otherwise, the value pointed to by *key*
 47542 shall be inserted (that is, a new node is created and the value of *key* is copied to this node), and a
 47543 pointer to this node returned. Only pointers are copied, so the application shall ensure that the
 47544 calling routine stores the data. The *rootp* argument points to a variable that points to the root
 47545 node of the tree. A null pointer value for the variable pointed to by *rootp* denotes an empty tree;
 47546 in this case, the variable shall be set to point to the node which shall be at the root of the new
 47547 tree.

47548 Like *tsearch()*, *tfind()* shall search for a node in the tree, returning a pointer to it if found.
 47549 However, if it is not found, *tfind()* shall return a null pointer. The arguments for *tfind()* are the
 47550 same as for *tsearch()*.

47551 The *tdelete()* function shall delete a node from a binary search tree. The arguments are the same
 47552 as for *tsearch()*. The variable pointed to by *rootp* shall be changed if the deleted node was the
 47553 root of the tree. The *tdelete()* function shall return a pointer to the parent of the deleted node, or a
 47554 null pointer if the node is not found.

47555 The *twalk()* function shall traverse a binary search tree. The *root* argument is a pointer to the root
 47556 node of the tree to be traversed. (Any node in a tree may be used as the root for a walk below
 47557 that node.) The argument *action* is the name of a routine to be invoked at each node. This routine
 47558 is, in turn, called with three arguments. The first argument shall be the address of the node being
 47559 visited. The structure pointed to by this argument is unspecified and shall not be modified by
 47560 the application, but it shall be possible to cast a pointer-to-node into a pointer-to-pointer-to-
 47561 element to access the element stored in the node. The second argument shall be a value from an
 47562 enumeration data type:

```
47563     typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

47564 (defined in `<search.h>`), depending on whether this is the first, second, or third time that the
 47565 node is visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a
 47566 leaf. The third argument shall be the level of the node in the tree, with the root being level 0.

47567 If the calling function alters the pointer to the root, the result is undefined.

47568 RETURN VALUE

47569 If the node is found, both `tsearch()` and `tfind()` shall return a pointer to it. If not, `tfind()` shall
 47570 return a null pointer, and `tsearch()` shall return a pointer to the inserted item.

47571 A null pointer shall be returned by `tsearch()` if there is not enough space available to create a new
 47572 node.

47573 A null pointer shall be returned by `tdelete()`, `tfind()`, and `tsearch()` if `rootp` is a null pointer on
 47574 entry.

47575 The `tdelete()` function shall return a pointer to the parent of the deleted node, or a null pointer if
 47576 the node is not found.

47577 The `twalk()` function shall not return a value.

47578 ERRORS

47579 No errors are defined.

47580 EXAMPLES

47581 The following code reads in strings and stores structures containing a pointer to each string and
 47582 a count of its length. It then walks the tree, printing out the stored strings and their lengths in
 47583 alphabetical order.

```
47584 #include <search.h>
47585 #include <string.h>
47586 #include <stdio.h>
47587
47588 #define STRSZ     10000
47589 #define NODSZ     500
47590
47591 struct node {      /* Pointers to these are stored in the tree. */
47592     char    *string;
47593     int     length;
47594 };
47595
47596     char   string_space[STRSZ]; /* Space to store strings. */
47597     struct node nodes[NODSZ];   /* Nodes to store. */
47598     void   *root = NULL;        /* This points to the root. */
47599
47600     int main(int argc, char *argv[])
47601 {
47602     char   *strptr = string_space;
47603     struct node  *nodeptr = nodes;
47604     void   print_node(const void *, VISIT, int);
47605     int    i = 0, node_compare(const void *, const void *);
47606
47607     while (gets(strptr) != NULL && i++ < NODSZ)  {
47608         /* Set node. */
47609         nodeptr->string = strptr;
47610         nodeptr->length = strlen(strptr);
47611         /* Put node into the tree. */
47612         (void) tsearch((void *)nodeptr, (void **)&root,
47613                         node_compare);
```

```

47609             /* Adjust pointers, so we do not overwrite tree. */
47610             strptr += nodeptr->length + 1;
47611             nodeptr++;
47612         }
47613         twalk(root, print_node);
47614         return 0;
47615     }
47616
47617     /*
47618      * This routine compares two nodes, based on an
47619      * alphabetical ordering of the string field.
47620      */
47621     int
47622     node_compare(const void *node1, const void *node2)
47623     {
47624         return strcmp(((const struct node *) node1)->string,
47625                     ((const struct node *) node2)->string);
47626     }
47627
47628     /*
47629      * This routine prints out a node, the second time
47630      * twalk encounters it or if it is a leaf.
47631      */
47632     void
47633     print_node(const void *ptr, VISIT order, int level)
47634     {
47635         const struct node *p = *(const struct node **) ptr;
47636
47637         if (order == postorder || order == leaf) {
47638             (void) printf("string = %s, length = %d\n",
47639                         p->string, p->length);
47640         }
47641     }

```

47639 APPLICATION USAGE

The *root* argument to *twalk()* is one level of indirection less than the *rootp* arguments to *tdelete()* and *tsearch()*.

There are two nomenclatures used to refer to the order in which tree nodes are visited. The *tsearch()* function uses **preorder**, **postorder**, and **endorder** to refer respectively to visiting a node before any of its children, after its left child and before its right, and after both its children. The alternative nomenclature uses **preorder**, **inorder**, and **postorder** to refer to the same visits, which could result in some confusion over the meaning of **postorder**.

47647 RATIONALE

47648 None.

47649 FUTURE DIRECTIONS

47650 None.

47651 SEE ALSO

47652 *hcreate()*, *lsearch()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**search.h**>

47653 CHANGE HISTORY

47654 First released in Issue 1. Derived from Issue 1 of the SVID.

47655 Issue 5

47656 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.
47657

47658 Issue 6

47659 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

47660 The **restrict** keyword is added to the *tdelete()* prototype for alignment with the
47661 ISO/IEC 9899:1999 standard.

47662 NAME

47663 **telldir** — current location of a named directory stream

47664 SYNOPSIS

47665 XSI `#include <dirent.h>`

47666 `long telldir(DIR *dirp);`

47667

47668 DESCRIPTION

47669 The **telldir()** function shall obtain the current location associated with the directory stream specified by *dirp*.

47671 If the most recent operation on the directory stream was a **seekdir()**, the directory position returned from the **telldir()** shall be the same as that supplied as a *loc* argument for **seekdir()**.

47673 RETURN VALUE

47674 Upon successful completion, **telldir()** shall return the current location of the specified directory stream.

47676 ERRORS

47677 No errors are defined.

47678 EXAMPLES

47679 None.

47680 APPLICATION USAGE

47681 None.

47682 RATIONALE

47683 None.

47684 FUTURE DIRECTIONS

47685 None.

47686 SEE ALSO

47687 `opendir()`, `readdir()`, `seekdir()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<dirent.h>`

47688 CHANGE HISTORY

47689 First released in Issue 2.

47690 NAME

47691 **tempnam** — create a name for a temporary file

47692 SYNOPSIS

47693 XSI **#include <stdio.h>**

47694 **char *tempnam(const char *dir, const char *pfx);**

47695

47696 DESCRIPTION

47697 The *tempnam*() function shall generate a pathname that may be used for a temporary file.

47698 The *tempnam*() function allows the user to control the choice of a directory. The *dir* argument
47699 points to the name of the directory in which the file is to be created. If *dir* is a null pointer or
47700 points to a string which is not a name for an appropriate directory, the path prefix defined as
47701 P_tmpdir in the **<stdio.h>** header shall be used. If that directory is not accessible, an
47702 implementation-defined directory may be used.

47703 Many applications prefer their temporary files to have certain initial letter sequences in their
47704 names. The *pfx* argument should be used for this. This argument may be a null pointer or point
47705 to a string of up to five bytes to be used as the beginning of the filename.

47706 Some implementations of *tempnam*() may use *tmpnam*() internally. On such implementations, if
47707 called more than {TMP_MAX} times in a single process, the behavior is implementation-defined.

47708 RETURN VALUE

47709 Upon successful completion, *tempnam*() shall allocate space for a string, put the generated
47710 pathname in that space, and return a pointer to it. The pointer shall be suitable for use in a
47711 subsequent call to *free*(). Otherwise, it shall return a null pointer and set *errno* to indicate the
47712 error.

47713 ERRORS

47714 The *tempnam*() function shall fail if:

47715 [ENOMEM] Insufficient storage space is available.

47716 EXAMPLES**47717 Generating a Pathname**

47718 The following example generates a pathname for a temporary file in directory **/tmp**, with the
47719 prefix *file*. After the filename has been created, the call to *free*() deallocates the space used to
47720 store the filename.

```
47721 #include <stdio.h>
47722 #include <stdlib.h>
47723 ...
47724 char *directory = "/tmp";
47725 char *fileprefix = "file";
47726 char *file;
47727 file = tempnam(directory, fileprefix);
47728 free(file);
```

47729 APPLICATION USAGE

47730 This function only creates pathnames. It is the application's responsibility to create and remove
47731 the files. Between the time a pathname is created and the file is opened, it is possible for some
47732 other process to create a file with the same name. Applications may find *tmpfile*() more useful.

47733 RATIONALE

47734 None.

47735 FUTURE DIRECTIONS

47736 None.

47737 SEE ALSO

47738 *open()*, *free()*, *open()*, *tmpfile()*, *tmpnam()*, *unlink()*, the Base Definitions volume of
47739 IEEE Std 1003.1-2001, <stdio.h>

47740 CHANGE HISTORY

47741 First released in Issue 1. Derived from Issue 1 of the SVID.

47742 Issue 5

47743 The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
47744 previous issues.

47745 NAME

47746 **tfind** — search binary search tree

47747 SYNOPSIS

47748 XSI `#include <search.h>`

47749 `void *tfind(const void *key, void *const *rootp,`
47750 `int (*compar)(const void *, const void *));`

47751

47752 DESCRIPTION

47753 Refer to *tdelete()*.

47754 NAME

47755 tgamma, tgammaf, tgammal — compute gamma() function

47756 SYNOPSIS

```
47757        #include <math.h>
47758        double tgamma(double x);
47759        float tgammaf(float x);
47760        long double tgammal(long double x);
```

47761 DESCRIPTION

47762 CX 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.

47765 These functions shall compute the *gamma()* function of *x*.

47766 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.

47770 RETURN VALUE

47771 Upon successful completion, these functions shall return *Gamma(x)*.

47772 If *x* is a negative integer, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

47774 If the correct value would cause overflow, a range error shall occur and *tgamma()*, *tgammaf()*, and *tgammal()* shall return ±HUGE_VAL, ±HUGE_VALF, or ±HUGE_VALL, respectively, with the same sign as the correct value of the function.

47777 MX If *x* is NaN, a NaN shall be returned.

47778 If *x* is +Inf, *x* shall be returned.

47779 If *x* is ±0, a pole error shall occur, and *tgamma()*, *tgammaf()*, and *tgammal()* shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL, respectively.

47781 If *x* is -Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

47783 ERRORS

47784 These functions shall fail if:

47785 MX Domain Error The value of *x* is a negative integer, or *x* is -Inf.

47786 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.

47790 MX Pole Error The value of *x* is zero.

47791 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 divide-by-zero floating-point exception shall be raised.

47795	Range Error	The value overflows.
47796		If the integer expression (<code>math_errhandling & MATH_ERRNO</code>) is non-zero, then <code>errno</code> shall be set to [ERANGE]. If the integer expression (<code>math_errhandling & MATH_ERREXCEPT</code>) is non-zero, then the overflow floating-point exception shall be raised.
47797		
47798		
47799		

47800 EXAMPLES

47801 None.

47802 APPLICATION USAGE

47803 For IEEE Std 754-1985 **double**, overflow happens when $0 < x < 1/\text{DBL_MAX}$, and $171.7 < x$.

47804 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling & MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

47806 RATIONALE

47807 This function is named *tgamma()* in order to avoid conflicts with the historical *gamma()* and *lgamma()* functions.

47809 FUTURE DIRECTIONS

47810 It is possible that the error response for a negative integer argument may be changed to a pole error and a return value of $\pm\text{Inf}$.

47812 SEE ALSO

47813 *feclearexcept()*, *fetestexcept()*, *lgamma()*, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

47815 CHANGE HISTORY

47816 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

47817 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/65 is applied, correcting the third paragraph in the RETURN VALUE section. 1
47818

47819 NAME

47820 time — get time

47821 SYNOPSIS

47822 #include <time.h>

47823 time_t time(time_t *tloc);

47824 DESCRIPTION

47825 CX 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.

47828 CX The *time()* function shall return the value of time in seconds since the Epoch.

47829 The *tloc* argument points to an area where the return value is also stored. If *tloc* is a null pointer, no value is stored.

47831 RETURN VALUE

47832 Upon successful completion, *time()* shall return the value of time. Otherwise, *(time_t)*-1 shall be returned.

47834 ERRORS

47835 No errors are defined.

47836 EXAMPLES

47837 Getting the Current Time

47838 The following example uses the *time()* function to calculate the time elapsed, in seconds, since
47839 the Epoch, *localtime()* to convert that value to a broken-down time, and *asctime()* to convert the
47840 broken-down time values into a printable string.

```
47841 #include <stdio.h>
47842 #include <time.h>
47843 int main(void)
47844 {
47845     time_t result;
47846
47847     result = time(NULL);
47848     printf("%s%ju secs since the Epoch\n",
47849             asctime(localtime(&result)),
47850             (uintmax_t)result);
47851     return(0);
47852 }
```

47853 This example writes the current time to *stdout* in a form like this:

```
47854
47855 Wed Jun 26 10:32:15 1996
47856 835810335 secs since the Epoch
```

47855 **Timing an Event**

47856 The following example gets the current time, prints it out in the user's format, and prints the
47857 number of minutes to an event being timed.

```
47858 #include <time.h>
47859 #include <stdio.h>
47860 ...
47861 time_t now;
47862 int minutes_to_event;
47863 ...
47864 time(&now);
47865 minutes_to_event = ...;
47866 printf("The time is ");
47867 puts(asctime(localtime(&now)));
47868 printf("There are %d minutes to the event.\n",
47869       minutes_to_event);
47870 ...
```

47871 **APPLICATION USAGE**

47872 None.

47873 **RATIONALE**

47874 The *time*() function returns a value in seconds (type **time_t**) while *times*() returns a set of values
47875 in clock ticks (type **clock_t**). Some historical implementations, such as 4.3 BSD, have
47876 mechanisms capable of returning more precise times (see below). A generalized timing scheme
47877 to unify these various timing mechanisms has been proposed but not adopted.

47878 Implementations in which **time_t** is a 32-bit signed integer (many historical implementations)
47879 fail in the year 2038. IEEE Std 1003.1-2001 does not address this problem. However, the use of
47880 the **time_t** type is mandated in order to ease the eventual fix.

47881 The use of the <**time.h**> header instead of <**sys/types.h**> allows compatibility with the ISO C
47882 standard.

47883 Many historical implementations (including Version 7) and the 1984 /usr/group standard use
47884 **long** instead of **time_t**. This volume of IEEE Std 1003.1-2001 uses the latter type in order to agree
47885 with the ISO C standard.

47886 4.3 BSD includes *time*() only as an alternate function to the more flexible *gettimeofday*() function.

47887 **FUTURE DIRECTIONS**

47888 In a future version of this volume of IEEE Std 1003.1-2001, **time_t** is likely to be required to be
47889 capable of representing times far in the future. Whether this will be mandated as a 64-bit type or
47890 a requirement that a specific date in the future be representable (for example, 10000 AD) is not
47891 yet determined. Systems purchased after the approval of this volume of IEEE Std 1003.1-2001
47892 should be evaluated to determine whether their lifetime will extend past 2038.

47893 **SEE ALSO**

47894 *asctime*(), *clock*(), *ctime*(), *difftime*(), *gettimeofday*(), *gmtime*(), *localtime*(), *mktme*(), *strftime*(),
47895 *strptime*(), *utime*(), the Base Definitions volume of IEEE Std 1003.1-2001, <**time.h**>

47896 **CHANGE HISTORY**

47897 First released in Issue 1. Derived from Issue 1 of the SVID.

47898 Issue 6

- 47899 Extensions beyond the ISO C standard are marked.
- 47900 The EXAMPLES, RATIONALE, and FUTURE DIRECTIONS sections are added.

47901 NAME

47902 timer_create — create a per-process timer (**REALTIME**)

47903 SYNOPSIS

```
47904 TMR #include <signal.h>
47905 #include <time.h>
47906 int timer_create(clockid_t clockid, struct sigevent *restrict evp,
47907     timer_t *restrict timerid);
```

47909 DESCRIPTION

47910 The *timer_create()* function shall create a per-process timer using the specified clock, *clock_id*, as
 47911 the timing base. The *timer_create()* function shall return, in the location referenced by *timerid*, a
 47912 timer ID of type **timer_t** used to identify the timer in timer requests. This timer ID shall be
 47913 unique within the calling process until the timer is deleted. The particular clock, *clock_id*, is
 47914 defined in **<time.h>**. The timer whose ID is returned shall be in a disarmed state upon return
 47915 from *timer_create()*.

47916 The *evp* argument, if non-NULL, points to a **sigevent** structure. This structure, allocated by the
 47917 application, defines the asynchronous notification to occur as specified in Section 2.4.1 (on page
 47918 28) when the timer expires. If the *evp* argument is NULL, the effect is as if the *evp* argument
 47919 pointed to a **sigevent** structure with the *sigev_notify* member having the value SIGEV_SIGNAL,
 47920 the *sigev_signo* having a default signal number, and the *sigev_value* member having the value of
 47921 the timer ID.

47922 Each implementation shall define a set of clocks that can be used as timing bases for per-process
 47923 MON timers. All implementations shall support a *clock_id* of CLOCK_REALTIME. If the Monotonic
 47924 Clock option is supported, implementations shall support a *clock_id* of CLOCK_MONOTONIC.

47925 Per-process timers shall not be inherited by a child process across a *fork()* and shall be disarmed
 47926 and deleted by an *exec*.

47927 CPT If **_POSIX_CPUTIME** is defined, implementations shall support *clock_id* values representing the
 47928 CPU-time clock of the calling process.

47929 TCT If **_POSIX_THREAD_CPUTIME** is defined, implementations shall support *clock_id* values
 47930 representing the CPU-time clock of the calling thread.

47931 CPT|TCT It is implementation-defined whether a *timer_create()* function will succeed if the value defined
 47932 by *clock_id* corresponds to the CPU-time clock of a process or thread different from the process
 47933 or thread invoking the function.

47934 TSA If *evp->sigev_notify* is SIGEV_THREAD and *sev->sigev_notify_attributes* is not NULL, if the 2
 47935 attribute pointed to by *sev->sigev_notify_attributes* has a thread stack address specified by a call 2
 47936 to *pthread_attr_setstack()* or *pthread_attr_setstackaddr()*, the results are unspecified if the signal is 2
 47937 generated more than once.

47938 RETURN VALUE

47939 If the call succeeds, *timer_create()* shall return zero and update the location referenced by *timerid*
 47940 to a **timer_t**, which can be passed to the per-process timer calls. If an error occurs, the function
 47941 shall return a value of -1 and set *errno* to indicate the error. The value of *timerid* is undefined if
 47942 an error occurs.

47943 ERRORS

47944 The *timer_create()* function shall fail if:

47945 [EAGAIN] The system lacks sufficient signal queuing resources to honor the request.

47946	[EAGAIN]	The calling process has already created all of the timers it is allowed by this implementation.
47947		
47948	[EINVAL]	The specified clock ID is not defined.

47949 CPT TCT	[ENOTSUP]	The implementation does not support the creation of a timer attached to the CPU-time clock that is specified by <i>clock_id</i> and associated with a process or thread different from the process or thread invoking <i>timer_create()</i> .
47950		
47951		

47952 EXAMPLES

47953 None.

47954 APPLICATION USAGE

47955 If a timer is created which has *evp->sigev_notify* set to SIGEV_THREAD and the attribute pointed to by *evp->sigev_notify_attributes* has a thread stack address specified by a call to either *pthread_attr_setstack()* or *pthread_attr_setstackaddr()*, the memory dedicated as a thread stack cannot be recovered. The reason for this is that the threads created in response to a timer expiration are created detached, or in an unspecified way if the thread attribute's *detachstate* is PTHREAD_CREATE_JOINABLE. In neither case is it valid to call *pthread_join()*, which makes it impossible to determine the lifetime of the created thread which thus means the stack memory cannot be reused. 2

47963 RATIONALE

47964 Periodic Timer Overrun and Resource Allocation

47965 The specified timer facilities may deliver realtime signals (that is, queued signals) on implementations that support this option. Since realtime applications cannot afford to lose notifications of asynchronous events, like timer expirations or asynchronous I/O completions, it must be possible to ensure that sufficient resources exist to deliver the signal when the event occurs. In general, this is not a difficulty because there is a one-to-one correspondence between a request and a subsequent signal generation. If the request cannot allocate the signal delivery resources, it can fail the call with an [EAGAIN] error. 2

47972 Periodic timers are a special case. A single request can generate an unspecified number of signals. This is not a problem if the requesting process can service the signals as fast as they are generated, thus making the signal delivery resources available for delivery of subsequent periodic timer expiration signals. But, in general, this cannot be assured—processing of periodic timer signals may “overrun”; that is, subsequent periodic timer expirations may occur before the currently pending signal has been delivered. 2

47978 Also, for signals, according to the POSIX.1-1990 standard, if subsequent occurrences of a pending signal are generated, it is implementation-defined whether a signal is delivered for each occurrence. This is not adequate for some realtime applications. So a mechanism is required to allow applications to detect how many timer expirations were delayed without requiring an indefinite amount of system resources to store the delayed expirations. 2

47983 The specified facilities provide for an overrun count. The overrun count is defined as the number of extra timer expirations that occurred between the time a timer expiration signal is generated and the time the signal is delivered. The signal-catching function, if it is concerned with overruns, can retrieve this count on entry. With this method, a periodic timer only needs one “signal queuing resource” that can be allocated at the time of the *timer_create()* function call. 2

47988 A function is defined to retrieve the overrun count so that an application need not allocate static storage to contain the count, and an implementation need not update this storage asynchronously on timer expirations. But, for some high-frequency periodic applications, the overhead of an additional system call on each timer expiration may be prohibitive. The 2

47992 functions, as defined, permit an implementation to maintain the overrun count in user space,
47993 associated with the *timerid*. The *timer_getoverrun()* function can then be implemented as a macro
47994 that uses the *timerid* argument (which may just be a pointer to a user space structure containing
47995 the counter) to locate the overrun count with no system call overhead. Other implementations,
47996 less concerned with this class of applications, can avoid the asynchronous update of user space
47997 by maintaining the count in a system structure at the cost of the extra system call to obtain it.

47998 Timer Expiration Signal Parameters

47999 The Realtime Signals Extension option supports an application-specific datum that is delivered
48000 to the extended signal handler. This value is explicitly specified by the application, along with
48001 the signal number to be delivered, in a **sigevent** structure. The type of the application-defined
48002 value can be either an integer constant or a pointer. This explicit specification of the value, as
48003 opposed to always sending the timer ID, was selected based on existing practice.

48004 It is common practice for realtime applications (on non-POSIX systems or realtime extended
48005 POSIX systems) to use the parameters of event handlers as the case label of a switch statement
48006 or as a pointer to an application-defined data structure. Since *timer_ids* are dynamically allocated
48007 by the *timer_create()* function, they can be used for neither of these functions without additional
48008 application overhead in the signal handler; for example, to search an array of saved timer IDs to
48009 associate the ID with a constant or application data structure.

48010 FUTURE DIRECTIONS

48011 None.

48012 SEE ALSO

48013 *clock_getres()*, *timer_delete()*, *timer_getoverrun()*, the Base Definitions volume of
48014 IEEE Std 1003.1-2001, <time.h>

48015 CHANGE HISTORY

48016 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

48017 Issue 6

48018 The *timer_create()* function is marked as part of the Timers option.

48019 The [ENOSYS] error condition has been removed as stubs need not be provided if an
48020 implementation does not support the Timers option.

48021 CPU-time clocks are added for alignment with IEEE Std 1003.1d-1999.

48022 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding the
48023 requirement for the CLOCK_MONOTONIC clock under the Monotonic Clock option.

48024 The **restrict** keyword is added to the *timer_create()* prototype for alignment with the
48025 ISO/IEC 9899:1999 standard.

48026 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/138 is applied, updating the
48027 DESCRIPTION and APPLICATION USAGE sections to describe the case when a timer is created
48028 with the notification method set to SIGEV_THREAD. 2 2 2

48029 **NAME**

48030 timer_delete — delete a per-process timer (**REALTIME**)

48031 **SYNOPSIS**

48032 TMR #include <time.h>

48033 int timer_delete(timer_t timerid);

48034

48035 **DESCRIPTION**

48036 The *timer_delete()* function deletes the specified timer, *timerid*, previously created by the
48037 *timer_create()* function. If the timer is armed when *timer_delete()* is called, the behavior shall be
48038 as if the timer is automatically disarmed before removal. The disposition of pending signals for
48039 the deleted timer is unspecified.

48040 **RETURN VALUE**

48041 If successful, the *timer_delete()* function shall return a value of zero. Otherwise, the function shall
48042 return a value of -1 and set *errno* to indicate the error.

48043 **ERRORS**

48044 The *timer_delete()* function may fail if:

2

48045 [EINVAL] The timer ID specified by *timerid* is not a valid timer ID.

48046 **EXAMPLES**

48047 None.

48048 **APPLICATION USAGE**

48049 None.

48050 **RATIONALE**

48051 None.

48052 **FUTURE DIRECTIONS**

48053 None.

48054 **SEE ALSO**

48055 *timer_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

48056 **CHANGE HISTORY**

48057 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

48058 **Issue 6**

48059 The *timer_delete()* function is marked as part of the Timers option.

48060 The [ENOSYS] error condition has been removed as stubs need not be provided if an
48061 implementation does not support the Timers option.

48062 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/139 is applied, updating the ERRORS 2
48063 section so that the [EINVAL] error becomes optional.

48064 NAME

48065 timer_getoverrun, timer_gettime, timer_settime — per-process timers (REALTIME)

48066 SYNOPSIS

```
48067 TMR #include <time.h>
48068 int timer_getoverrun(timer_t timerid);
48069 int timer_gettime(timer_t timerid, struct itimerspec *value);
48070 int timer_settime(timer_t timerid, int flags,
48071     const struct itimerspec *restrict value,
48072     struct itimerspec *restrict ovalue);
48073
```

48074 DESCRIPTION

48075 The *timer_gettime()* function shall store the amount of time until the specified timer, *timerid*,
 48076 expires and the reload value of the timer into the space pointed to by the *value* argument. The
 48077 *it_value* member of this structure shall contain the amount of time before the timer expires, or
 48078 zero if the timer is disarmed. This value is returned as the interval until timer expiration, even if
 48079 the timer was armed with absolute time. The *it_interval* member of *value* shall contain the reload
 48080 value last set by *timer_settime()*.

48081 The *timer_settime()* function shall set the time until the next expiration of the timer specified by
 48082 *timerid* from the *it_value* member of the *value* argument and arm the timer if the *it_value* member
 48083 of *value* is non-zero. If the specified timer was already armed when *timer_settime()* is called, this
 48084 call shall reset the time until next expiration to the *value* specified. If the *it_value* member of *value*
 48085 is zero, the timer shall be disarmed. The effect of disarming or resetting a timer with pending
 48086 expiration notifications is unspecified.

48087 If the flag TIMER_ABSTIME is not set in the argument *flags*, *timer_settime()* shall behave as if the
 48088 time until next expiration is set to be equal to the interval specified by the *it_value* member of
 48089 *value*. That is, the timer shall expire in *it_value* nanoseconds from when the call is made. If the
 48090 flag TIMER_ABSTIME is set in the argument *flags*, *timer_settime()* shall behave as if the time
 48091 until next expiration is set to be equal to the difference between the absolute time specified by
 48092 the *it_value* member of *value* and the current value of the clock associated with *timerid*. That is,
 48093 the timer shall expire when the clock reaches the value specified by the *it_value* member of *value*.
 48094 If the specified time has already passed, the function shall succeed and the expiration
 48095 notification shall be made.

48096 The reload value of the timer shall be set to the value specified by the *it_interval* member of
 48097 *value*. When a timer is armed with a non-zero *it_interval*, a periodic (or repetitive) timer is
 48098 specified.

48099 Time values that are between two consecutive non-negative integer multiples of the resolution
 48100 of the specified timer shall be rounded up to the larger multiple of the resolution. Quantization
 48101 error shall not cause the timer to expire earlier than the rounded time value.

48102 If the argument *ovalue* is not NULL, the *timer_settime()* function shall store, in the location
 48103 referenced by *ovalue*, a value representing the previous amount of time before the timer would
 48104 have expired, or zero if the timer was disarmed, together with the previous timer reload value.
 48105 Timers shall not expire before their scheduled time.

48106 Only a single signal shall be queued to the process for a given timer at any point in time. When a
 48107 timer for which a signal is still pending expires, no signal shall be queued, and a timer overrun
 48108 RTS shall occur. When a timer expiration signal is delivered to or accepted by a process, if the
 48109 implementation supports the Realtime Signals Extension, the *timer_getoverrun()* function shall
 48110 return the timer expiration overrun count for the specified timer. The overrun count returned
 48111 contains the number of extra timer expirations that occurred between the time the signal was

48112 generated (queued) and when it was delivered or accepted, up to but not including an
48113 implementation-defined maximum of {DELAYTIMER_MAX}. If the number of such extra
48114 expirations is greater than or equal to {DELAYTIMER_MAX}, then the overrun count shall be set
48115 to {DELAYTIMER_MAX}. The value returned by *timer_getoverrun()* shall apply to the most
48116 recent expiration signal delivery or acceptance for the timer. If no expiration signal has been
48117 delivered for the timer, or if the Realtime Signals Extension is not supported, the return value of
48118 *timer_getoverrun()* is unspecified.

48119 RETURN VALUE

48120 If the *timer_getoverrun()* function succeeds, it shall return the timer expiration overrun count as
48121 explained above.

48122 If the *timer_gettime()* or *timer_settime()* functions succeed, a value of 0 shall be returned.

48123 If an error occurs for any of these functions, the value -1 shall be returned, and *errno* set to
48124 indicate the error.

48125 ERRORS

48126 The *timer_settime()* function shall fail if:

2

48127 [EINVAL] A *value* structure specified a nanosecond value less than zero or greater than
48128 or equal to 1 000 million, and the *it_value* member of that structure did not
48129 specify zero seconds and nanoseconds.

2 2 2

48130 These functions may fail if:

2

48131 [EINVAL] The *timerid* argument does not correspond to an ID returned by *timer_create()*
48132 but not yet deleted by *timer_delete()*.

2

48133 The *timer_settime()* function may fail if:

2

48134 [EINVAL] The *it_interval* member of *value* is not zero and the timer was created with
48135 notification by creation of a new thread (*sigev_sigev_notify* was
48136 SIGEV_THREAD) and a fixed stack address has been set in the thread
48137 attribute pointed to by *sigev_notify_attributes*.

2 2 2 2

48138 EXAMPLES

48139 None.

48140 APPLICATION USAGE

48141 Using fixed stack addresses is problematic when timer expiration is signalled by the creation of a
48142 new thread. Since it cannot be assumed that the thread created for one expiration is finished
48143 before the next expiration of the timer, it could happen that two threads use the same memory
48144 as a stack at the same time. This is invalid and produces undefined results.

2 2 2 2

48145 RATIONALE

48146 Practical clocks tick at a finite rate, with rates of 100 hertz and 1 000 hertz being common. The
48147 inverse of this tick rate is the clock resolution, also called the clock granularity, which in either
48148 case is expressed as a time duration, being 10 milliseconds and 1 millisecond respectively for
48149 these common rates. The granularity of practical clocks implies that if one reads a given clock
48150 twice in rapid succession, one may get the same time value twice; and that timers must wait for
48151 the next clock tick after the theoretical expiration time, to ensure that a timer never returns too
48152 soon. Note also that the granularity of the clock may be significantly coarser than the resolution
48153 of the data format used to set and get time and interval values. Also note that some
48154 implementations may choose to adjust time and/or interval values to exactly match the ticks of
48155 the underlying clock.

48156 This volume of IEEE Std 1003.1-2001 defines functions that allow an application to determine the
48157 implementation-supported resolution for the clocks and requires an implementation to

48158 document the resolution supported for timers and *nanosleep()* if they differ from the supported
48159 clock resolution. This is more of a procurement issue than a runtime application issue.

48160 FUTURE DIRECTIONS

48161 None.

48162 SEE ALSO

48163 *clock_getres()*, *timer_create()*, the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

48164 CHANGE HISTORY

48165 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

48166 Issue 6

48167 The *timer_getoverrun()*, *timer_gettime()*, and *timer_settime()* functions are marked as part of the
48168 Timers option.

48169 The [ENOSYS] error condition has been removed as stubs need not be provided if an
48170 implementation does not support the Timers option.

48171 The [EINVAL] error condition is updated to include the following: “and the *it_value* member of
48172 that structure did not specify zero seconds and nanoseconds.” This change is for IEEE PASC
48173 Interpretation 1003.1 #89.

48174 The DESCRIPTION for *timer_getoverrun()* is updated to clarify that “If no expiration signal has
48175 been delivered for the timer, or if the Realtime Signals Extension is not supported, the return
48176 value of *timer_getoverrun()* is unspecified”.

48177 The **restrict** keyword is added to the *timer_settime()* prototype for alignment with the
48178 ISO/IEC 9899:1999 standard.

48179 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/140 is applied, updating the ERRORS 2
48180 section so that the mandatory [EINVAL] error (“The *timerid* argument does not correspond to an 2
48181 ID returned by *timer_create()* but not yet deleted by *timer_delete()*”) becomes optional. 2

48182 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/141 is applied, updating the ERRORS 2
48183 section to include an optional [EINVAL] error for the case when a timer is created with the 2
48184 notification method set to SIGEV_THREAD. APPLICATION USAGE text is also added. 2

48185 NAME

48186 times — get process and waited-for child process times

48187 SYNOPSIS

```
48188 #include <sys/times.h>
48189 clock_t times(struct tms *buffer);
```

48190 DESCRIPTION

48191 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>**.

48193 All times are measured in terms of the number of clock ticks used.

48194 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 48195 a child process has not waited for its children, their times shall not be included in its times.

- 48196
- The *tms_utime* structure member is the CPU time charged for the execution of user instructions of the calling process.
 - The *tms_stime* structure member is the CPU time charged for execution by the system on behalf of the calling process.
 - The *tms_cutime* structure member is the sum of the *tms_utime* and *tms_cutime* times of the child processes.
 - The *tms_cstime* structure member is the sum of the *tms_stime* and *tms_cstime* times of the child processes.

48205 RETURN VALUE

48206 Upon successful completion, *times()* shall return the elapsed real time, in clock ticks, since an 48207 arbitrary point in the past (for example, system start-up time). This point does not change from 48208 one invocation of *times()* within the process to another. The return value may overflow the 48209 possible range of type **clock_t**. If *times()* fails, (**clock_t**)−1 shall be returned and *errno* set to 48210 indicate the error.

48211 ERRORS

48212 No errors are defined.

48213 EXAMPLES

48214 Timing a Database Lookup

48215 The following example defines two functions, *start_clock()* and *end_clock()*, that are used to time 48216 a lookup. It also defines variables of type **clock_t** and **tms** to measure the duration of 48217 transactions. The *start_clock()* function saves the beginning times given by the *times()* function. 48218 The *end_clock()* function gets the ending times and prints the difference between the two times.

```
48219 #include <sys/times.h>
48220 #include <stdio.h>
48221 ...
48222 void start_clock(void);
48223 void end_clock(char *msg);
48224 ...
48225 static clock_t st_time;
48226 static clock_t en_time;
48227 static struct tms st_cpu;
48228 static struct tms en_cpu;
```

```
48229     ...
48230     void
48231     start_clock()
48232     {
48233         st_time = times(&st_cpu);
48234     }
48235
48236     /* This example assumes that the result of each subtraction
48237     is within the range of values that can be represented in
48238     an integer type. */
48239     void
48240     end_clock(char *msg)
48241     {
48242         en_time = times(&en_cpu);
48243
48244         fputs(msg,stdout);
48245         printf("Real Time: %jd, User Time %jd, System Time %jd\n",
48246                (intmax_t)(en_time - st_time),
48247                (intmax_t)(en_cpu.tms_utime - st_cpu.tms_utime),
48248                (intmax_t)(en_cpu.tms_stime - st_cpu.tms_stime));
48249     }
```

48248 APPLICATION USAGE

48249 Applications should use *sysconf(_SC_CLK_TCK)* to determine the number of clock ticks per
48250 second as it may vary from system to system.

48251 RATIONALE

48252 The accuracy of the times reported is intentionally left unspecified to allow implementations
48253 flexibility in design, from uniprocessor to multi-processor networks.

48254 The inclusion of times of child processes is recursive, so that a parent process may collect the
48255 total times of all of its descendants. But the times of a child are only added to those of its parent
48256 when its parent successfully waits on the child. Thus, it is not guaranteed that a parent process
48257 can always see the total times of all its descendants; see also the discussion of the term
48258 “realtime” in *alarm()*.

48259 If the type **clock_t** is defined to be a signed 32-bit integer, it overflows in somewhat more than a
48260 year if there are 60 clock ticks per second, or less than a year if there are 100. There are individual
48261 systems that run continuously for longer than that. This volume of IEEE Std 1003.1-2001 permits
48262 an implementation to make the reference point for the returned value be the start-up time of the
48263 process, rather than system start-up time.

48264 The term “charge” in this context has nothing to do with billing for services. The operating
48265 system accounts for time used in this way. That information must be correct, regardless of how
48266 that information is used.

48267 FUTURE DIRECTIONS

48268 None.

48269 SEE ALSO

48270 *alarm()*, *exec*, *fork()*, *sysconf()*, *time()*, *wait()*, the Base Definitions volume of
48271 IEEE Std 1003.1-2001, <*sys/times.h*>

48272 CHANGE HISTORY

48273 First released in Issue 1. Derived from Issue 1 of the SVID.

48274 NAME

48275 timezone — difference from UTC and local standard time

48276 SYNOPSIS

48277 XSI #include <time.h>

48278 extern long timezone;

48279

48280 DESCRIPTION

48281 Refer to *tzset()*.

48282 NAME

48283 tmpfile — create a temporary file

48284 SYNOPSIS

```
48285 #include <stdio.h>
48286 FILE *tmpfile(void);
```

48287 DESCRIPTION

48288 CX 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.

48291 The *tmpfile()* function shall create a temporary file and open a corresponding stream. The file shall be automatically deleted when all references to the file are closed. The file is opened as in *open()* for update (w+).

48294 CX In some implementations, a permanent file may be left behind if the process calling *tmpfile()* is killed while it is processing a call to *tmpfile()*.

48296 An error message may be written to standard error if the stream cannot be opened.

48297 RETURN VALUE

48298 Upon successful completion, *tmpfile()* shall return a pointer to the stream of the file that is created. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

48300 ERRORS

48301 The *tmpfile()* function shall fail if:

48302 CX [EINTR] A signal was caught during *tmpfile()*.
48303 CX [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.
48304 CX [ENFILE] The maximum allowable number of files is currently open in the system.
48305 CX [ENOSPC] The directory or file system which would contain the new file cannot be expanded.
48307 CX [EOVERFLOW] The file is a regular file and the size of the file cannot be represented correctly in an object of type *off_t*.

48309 The *tmpfile()* function may fail if:

48310 CX [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.
48311 CX [ENOMEM] Insufficient storage space is available.

48312 EXAMPLES

48313 Creating a Temporary File

48314 The following example creates a temporary file for update, and returns a pointer to a stream for
48315 the created file in the *fp* variable.

```
48316 #include <stdio.h>
48317 ...
48318 FILE *fp;
48319 fp = tmpfile();
```

48320 APPLICATION USAGE

48321 It should be possible to open at least {TMP_MAX} temporary files during the lifetime of the
48322 program (this limit may be shared with *tmpnam()*) and there should be no limit on the number
48323 simultaneously open other than this limit and any limit on the number of open files
48324 ({FOPEN_MAX}).

48325 RATIONALE

48326 None.

48327 FUTURE DIRECTIONS

48328 None.

48329 SEE ALSO

48330 *open()*, *tmpnam()*, *unlink()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

48331 CHANGE HISTORY

48332 First released in Issue 1. Derived from Issue 1 of the SVID.

48333 Issue 5

48334 Large File Summit extensions are added.

48335 The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes
48336 in previous issues.

48337 Issue 6

48338 Extensions beyond the ISO C standard are marked.

48339 The following new requirements on POSIX implementations derive from alignment with the
48340 Single UNIX Specification:

- 48341 • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support
48342 large files.
- 48343 • The [EMFILE] optional error condition is added.

48344 The APPLICATION USAGE section is added for alignment with the ISO/IEC 9899:1999
48345 standard.

48346 NAME

48347 tmpnam — create a name for a temporary file

48348 SYNOPSIS

```
48349       #include <stdio.h>
48350       char *tmpnam(char *s);
```

48351 DESCRIPTION

48352 CX 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.

48355 The *tmpnam()* function shall generate a string that is a valid filename and that is not the same as the name of an existing file. The function is potentially capable of generating {TMP_MAX} different strings, but any or all of them may already be in use by existing files and thus not be suitable return values.

48359 The *tmpnam()* function generates a different string each time it is called from the same process, up to {TMP_MAX} times. If it is called more than {TMP_MAX} times, the behavior is implementation-defined.

48362 The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-2001, except *tempnam()*, calls *tmpnam()*. 2

48364 CX If the application uses any of the functions guaranteed to be available if either _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS is defined, the application shall ensure that the *tmpnam()* function is called with a non-NUL parameter.

48367 RETURN VALUE

48368 Upon successful completion, *tmpnam()* shall return a pointer to a string. If no suitable string can be generated, the *tmpnam()* function shall return a null pointer.

48370 If the argument *s* is a null pointer, *tmpnam()* shall leave its result in an internal static object and return a pointer to that object. Subsequent calls to *tmpnam()* may modify the same object. If the argument *s* is not a null pointer, it is presumed to point to an array of at least L_tmpnam chars; *tmpnam()* shall write its result in that array and shall return the argument as its value.

48374 ERRORS

48375 No errors are defined.

48376 EXAMPLES

48377 Generating a Filename

48378 The following example generates a unique filename and stores it in the array pointed to by *ptr*.

```
48379       #include <stdio.h>
48380       ...
48381       char filename[L_tmpnam+1];
48382       char *ptr;
48383
48384       ptr = tmpnam(filename);
```

48384 APPLICATION USAGE

48385 This function only creates filenames. It is the application's responsibility to create and remove the files.

48387 Between the time a pathname is created and the file is opened, it is possible for some other process to create a file with the same name. Applications may find *tmpfile()* more useful.

48389 **RATIONALE**

48390 None.

48391 **FUTURE DIRECTIONS**

48392 None.

48393 **SEE ALSO**

48394 *open()*, *open()*, *tempnam()*, *tmpfile()*, *unlink()*, the Base Definitions volume of
48395 IEEE Std 1003.1-2001, <stdio.h>

48396 **CHANGE HISTORY**

48397 First released in Issue 1. Derived from Issue 1 of the SVID.

48398 **Issue 5**

48399 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

48400 **Issue 6**

48401 Extensions beyond the ISO C standard are marked.

48402 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

48403 The DESCRIPTION is expanded for alignment with the ISO/IEC 9899:1999 standard.

48404 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/142 is applied, updating the 2
48405 DESCRIPTION to allow implementations of the *tempnam()* function to call *tmpnam()*. 2

48406 NAME

48407 *toascii* — translate an integer to a 7-bit ASCII character

48408 SYNOPSIS

48409 XSI

```
#include <ctype.h>
```

48410

```
int toascii(int c);
```

48411

48412 DESCRIPTION

48413 The *toascii()* function shall convert its argument into a 7-bit ASCII character.

48414 RETURN VALUE

48415 The *toascii()* function shall return the value (*c* &0x7f).

48416 ERRORS

48417 No errors are returned.

48418 EXAMPLES

48419 None.

48420 APPLICATION USAGE

48421 None.

48422 RATIONALE

48423 None.

48424 FUTURE DIRECTIONS

48425 None.

48426 SEE ALSO

48427 *isascii()*, the Base Definitions volume of IEEE Std 1003.1-2001, *<ctype.h>*

48428 CHANGE HISTORY

48429 First released in Issue 1. Derived from Issue 1 of the SVID.

48430 NAME

48431 `tolower` — transliterate uppercase characters to lowercase

48432 SYNOPSIS

```
48433        #include <ctype.h>
48434        int tolower(int c);
```

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 `tolower()` function has as a domain a type `int`, the value of which is representable as an
48440 `unsigned char` or the value of EOF. If the argument has any other value, the behavior is
48441 undefined. If the argument of `tolower()` represents an uppercase letter, and there exists a
48442 CX corresponding lowercase letter (as defined by character type information in the program locale
48443 category `LC_CTYPE`), the result shall be the corresponding lowercase letter. All other arguments
48444 in the domain are returned unchanged.

48445 RETURN VALUE

48446 Upon successful completion, `tolower()` shall return the lowercase letter corresponding to the
48447 argument passed; otherwise, it shall return the argument unchanged.

48448 ERRORS

48449 No errors are defined.

48450 EXAMPLES

48451 None.

48452 APPLICATION USAGE

48453 None.

48454 RATIONALE

48455 None.

48456 FUTURE DIRECTIONS

48457 None.

48458 SEE ALSO

48459 `setlocale()`, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, `<ctype.h>`

48460 CHANGE HISTORY

48461 First released in Issue 1. Derived from Issue 1 of the SVID.

48462 Issue 6

48463 Extensions beyond the ISO C standard are marked.

48464 NAME

48465 **toupper** — transliterate lowercase characters to uppercase

48466 SYNOPSIS

```
48467       #include <ctype.h>
48468       int toupper(int c);
```

48469 DESCRIPTION

48470 CX The functionality described on this reference page is aligned with the ISO C standard. Any
48471 conflict between the requirements described here and the ISO C standard is unintentional. This
48472 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

48473 The *toupper()* function has as a domain a type **int**, the value of which is representable as an
48474 **unsigned char** or the value of EOF. If the argument has any other value, the behavior is
48475 undefined. If the argument of *toupper()* represents a lowercase letter, and there exists a
48476 CX corresponding uppercase letter (as defined by character type information in the program locale
48477 category *LC_CTYPE*), the result shall be the corresponding uppercase letter. All other arguments
48478 in the domain are returned unchanged.

48479 RETURN VALUE

48480 Upon successful completion, *toupper()* shall return the uppercase letter corresponding to the
48481 argument passed.

48482 ERRORS

48483 No errors are defined.

48484 EXAMPLES

48485 None.

48486 APPLICATION USAGE

48487 None.

48488 RATIONALE

48489 None.

48490 FUTURE DIRECTIONS

48491 None.

48492 SEE ALSO

48493 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

48494 CHANGE HISTORY

48495 First released in Issue 1. Derived from Issue 1 of the SVID.

48496 Issue 6

48497 Extensions beyond the ISO C standard are marked.

48498 NAME

48499 towctrans — wide-character transliteration

48500 SYNOPSIS

```
48501 #include <wctype.h>
48502 wint_t towctrans(wint_t wc, wctrans_t desc);
```

48503 DESCRIPTION

48504 CX 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.

48507 The *towctrans()* function shall transliterate the wide-character code *wc* using the mapping described by *desc*. The current setting of the *LC_CTYPE* category should be the same as during 48508 the call to *wctrans()* that returned the value *desc*. If the value of *desc* is invalid (that is, not 48509 obtained by a call to *wctrans()* or *desc* is invalidated by a subsequent call to *setlocale()* that has 48510 affected category *LC_CTYPE*), the result is unspecified.

48512 An application wishing to check for error situations should set *errno* to 0 before calling 48513 *towctrans()*. If *errno* is non-zero on return, an error has occurred.

48514 RETURN VALUE

48515 If successful, the *towctrans()* function shall return the mapped value of *wc* using the mapping 48516 described by *desc*. Otherwise, it shall return *wc* unchanged.

48517 ERRORS

48518 The *towctrans()* function may fail if:

48519 CX [EINVAL] *desc* contains an invalid transliteration descriptor.

48520 EXAMPLES

48521 None.

48522 APPLICATION USAGE

48523 The strings "tolower" and "toupper" are reserved for the standard mapping names. In the 48524 table below, the functions in the left column are equivalent to the functions in the right column.

48525 towlower(<i>wc</i>)	48526 towctrans(<i>wc</i> , wctrans("tolower"))
48526 towupper(<i>wc</i>)	48527 towctrans(<i>wc</i> , wctrans("toupper"))

48527 RATIONALE

48528 None.

48529 FUTURE DIRECTIONS

48530 None.

48531 SEE ALSO

48532 *tolower()*, *toupper()*, *wctrans()*, the Base Definitions volume of IEEE Std 1003.1-2001,
48533 <wctype.h>

48534 CHANGE HISTORY

48535 First released in Issue 5. Derived from ISO/IEC 9899:1990/Amendment 1:1995 (E).

48536 Issue 6

48537 Extensions beyond the ISO C standard are marked.

48538 NAME

48539 *towlower* — transliterate uppercase wide-character code to lowercase

48540 SYNOPSIS

```
48541        #include <wctype.h>
48542        wint_t towlower(wint_t wc);
```

48543 DESCRIPTION

48544 CX 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.

48547 The *towlower()* function has as a domain a type *wint_t*, the value of which the application shall ensure is a character representable as a *wchar_t*, and a wide-character code corresponding to a valid character in the current locale or the value of WEOF. If the argument has any other value, the behavior is undefined. If the argument of *towlower()* represents an uppercase wide-character code, and there exists a corresponding lowercase wide-character code (as defined by character type information in the program locale category *LC_CTYPE*), the result shall be the corresponding lowercase wide-character code. All other arguments in the domain are returned unchanged.

48555 RETURN VALUE

48556 Upon successful completion, *towlower()* shall return the lowercase letter corresponding to the argument passed; otherwise, it shall return the argument unchanged.

48558 ERRORS

48559 No errors are defined.

48560 EXAMPLES

48561 None.

48562 APPLICATION USAGE

48563 None.

48564 RATIONALE

48565 None.

48566 FUTURE DIRECTIONS

48567 None.

48568 SEE ALSO

48569 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, *<wctype.h>*, *<wchar.h>*

48571 CHANGE HISTORY

48572 First released in Issue 4.

48573 Issue 5

48574 The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the *<wctype.h>* header rather than *<wchar.h>*.

48578 Issue 6

48579 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

48580 NAME

48581 *towupper* — transliterate lowercase wide-character code to uppercase

48582 SYNOPSIS

```
48583        #include <wctype.h>
48584        wint_t towupper(wint_t wc);
```

48585 DESCRIPTION

48586 CX 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.

48589 The *towupper()* function has as a domain a type **wint_t**, the value of which the application shall ensure is a character representable as a **wchar_t**, and a wide-character code corresponding to a valid character in the current locale or the value of WEOF. If the argument has any other value, the behavior is undefined. If the argument of *towupper()* represents a lowercase wide-character code, and there exists a corresponding uppercase wide-character code (as defined by character type information in the program locale category *LC_CTYPE*), the result shall be the corresponding uppercase wide-character code. All other arguments in the domain are returned unchanged.

48597 RETURN VALUE

48598 Upon successful completion, *towupper()* shall return the uppercase letter corresponding to the argument passed. Otherwise, it shall return the argument unchanged.

48600 ERRORS

48601 No errors are defined.

48602 EXAMPLES

48603 None.

48604 APPLICATION USAGE

48605 None.

48606 RATIONALE

48607 None.

48608 FUTURE DIRECTIONS

48609 None.

48610 SEE ALSO

48611 *setlocale()*, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, **<wctype.h>**, **<wchar.h>**

48613 CHANGE HISTORY

48614 First released in Issue 4.

48615 Issue 5

48616 The following change has been made in this issue for alignment with
48617 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 48618 • The SYNOPSIS has been changed to indicate that this function and associated data types are
48619 now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

48620 Issue 6

48621 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

48622 NAME

48623

48624 SYNOPSIS

```
48625       #include <math.h>
48626       double trunc(double x);
48627       float truncf(float x);
48628       long double truncl(long double x);
```

48629 DESCRIPTION

48630 CX 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.

48633 These functions shall round their argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument.

48635 RETURN VALUE

48636 Upon successful completion, these functions shall return the truncated integer value.

48637 MX If x is NaN, a NaN shall be returned.

48638 If x is ± 0 or $\pm \text{Inf}$, x shall be returned.

48639 ERRORS

48640 No errors are defined.

48641 EXAMPLES

48642 None.

48643 APPLICATION USAGE

48644 None.

48645 RATIONALE

48646 None.

48647 FUTURE DIRECTIONS

48648 None.

48649 SEE ALSO

48650 The Base Definitions volume of IEEE Std 1003.1-2001, <**math.h**>

48651 CHANGE HISTORY

48652 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

48653 NAME

48654 truncate — truncate a file to a specified length

48655 SYNOPSIS

48656 XSI #include <unistd.h>

```
48657       int truncate(const char *path, off_t length);
```

48658

48659 DESCRIPTION

48660 The *truncate()* function shall cause the regular file named by *path* to have a size which shall be
48661 equal to *length* bytes.

48662 If the file previously was larger than *length*, the extra data is discarded. If the file was previously
48663 shorter than *length*, its size is increased, and the extended area appears as if it were zero-filled.

48664 The application shall ensure that the process has write permission for the file.

48665 If the request would cause the file size to exceed the soft file size limit for the process, the
48666 request shall fail and the implementation shall generate the SIGXFSZ signal for the process.

48667 This function shall not modify the file offset for any open file descriptions associated with the
48668 file. Upon successful completion, if the file size is changed, this function shall mark for update
48669 the *st_ctime* and *st_mtime* fields of the file, and the S_ISUID and S_ISGID bits of the file mode
48670 may be cleared.

48671 RETURN VALUE

48672 Upon successful completion, *truncate()* shall return 0. Otherwise, -1 shall be returned, and *errno*
48673 set to indicate the error.

48674 ERRORS

48675 The *truncate()* function shall fail if:

48676 [EINTR] A signal was caught during execution.

48677 [EINVAL] The *length* argument was less than 0.

48678 [EFBIG] or [EINVAL]

48679 The *length* argument was greater than the maximum file size.

48680 [EIO] An I/O error occurred while reading from or writing to a file system.

48681 [EACCES] A component of the path prefix denies search permission, or write permission
48682 is denied on the file.

48683 [EISDIR] The named file is a directory.

48684 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
48685 argument.

48686 [ENAMETOOLONG]

48687 The length of the *path* argument exceeds {PATH_MAX} or a pathname
48688 component is longer than {NAME_MAX}.

48689 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

48690 [ENOTDIR] A component of the path prefix of *path* is not a directory.

48691 [EROFS] The named file resides on a read-only file system.

48692 The *truncate()* function may fail if:

48693 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
48694 resolution of the *path* argument.

48695 [ENAMETOOLONG]
48696 Pathname resolution of a symbolic link produced an intermediate result
48697 whose length exceeds {PATH_MAX}.

48698 EXAMPLES

48699 None.

48700 APPLICATION USAGE

48701 None.

48702 RATIONALE

48703 None.

48704 FUTURE DIRECTIONS

48705 None.

48706 SEE ALSO

48707 *open()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

48708 CHANGE HISTORY

48709 First released in Issue 4, Version 2.

48710 Issue 5

48711 Moved from X/OPEN UNIX extension to BASE.

48712 Large File Summit extensions are added.

48713 Issue 6

48714 This reference page is split out from the *ftruncate()* reference page.

48715 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

48716 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
48717 [ELOOP] error condition is added.

48718 NAME

48719 truncf, truncI — round to truncated integer value

48720 SYNOPSIS

48721 #include <math.h>

48722 float truncf(float x);

48723 long double truncI(long double x);

48724 DESCRIPTION

48725 Refer to *trunc()*.

48726 **NAME**48727 **tsearch** — search a binary search tree48728 **SYNOPSIS**

```
48729 XSI #include <search.h>
48730     void *tsearch(const void *key, void **rootp,
48731             int (*compar)(const void *, const void *));
48732
```

48733 **DESCRIPTION**48734 Refer to *tdelete()*.

48735 NAME

48736 ttynname, ttynname_r — find the pathname of a terminal

48737 SYNOPSIS

```
48738 #include <unistd.h>
48739 char *ttynname(int fildes);
48740 TSF int ttynname_r(int fildes, char *name, size_t namesize);
48741
```

48742 DESCRIPTION

48743 The *ttynname()* function shall return a pointer to a string containing a null-terminated pathname
48744 of the terminal associated with file descriptor *fildes*. The return value may point to static data
48745 whose content is overwritten by each call.

48746 The *ttynname()* function need not be reentrant. A function that is not required to be reentrant is
48747 not required to be thread-safe.

48748 TSF The *ttynname_r()* function shall store the null-terminated pathname of the terminal associated
48749 with the file descriptor *fildes* in the character array referenced by *name*. The array is *namesize*
48750 characters long and should have space for the name and the terminating null character. The
48751 maximum length of the terminal name shall be {TTY_NAME_MAX}.

48752 RETURN VALUE

48753 Upon successful completion, *ttynname()* shall return a pointer to a string. Otherwise, a null
48754 pointer shall be returned and *errno* set to indicate the error.

48755 TSF If successful, the *ttynname_r()* function shall return zero. Otherwise, an error number shall be
48756 returned to indicate the error.

48757 ERRORS

48758 The *ttynname()* function may fail if:

48759 [EBADF] The *fildes* argument is not a valid file descriptor.

48760 [ENOTTY] The *fildes* argument does not refer to a terminal.

48761 The *ttynname_r()* function may fail if:

48762 TSF [EBADF] The *fildes* argument is not a valid file descriptor.

48763 TSF [ENOTTY] The *fildes* argument does not refer to a terminal.

48764 TSF [ERANGE] The value of *namesize* is smaller than the length of the string to be returned
48765 including the terminating null character.

48766 EXAMPLES

48767 None.

48768 APPLICATION USAGE

48769 None.

48770 RATIONALE

48771 The term “terminal” is used instead of the historical term “terminal device” in order to avoid a
48772 reference to an undefined term.

48773 The thread-safe version places the terminal name in a user-supplied buffer and returns a non-
48774 zero value if it fails. The non-thread-safe version may return the name in a static data area that
48775 may be overwritten by each call.

48776 FUTURE DIRECTIONS

48777 None.

48778 SEE ALSO

48779 The Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

48780 CHANGE HISTORY

48781 First released in Issue 1. Derived from Issue 1 of the SVID.

48782 Issue 5

48783 The *ttyname_r()* function is included for alignment with the POSIX Threads Extension.

48784 A note indicating that the *ttyname()* function need not be reentrant is added to the
48785 DESCRIPTION.

48786 Issue 6

48787 The *ttyname_r()* function is marked as part of the Thread-Safe Functions option.

48788 The following new requirements on POSIX implementations derive from alignment with the
48789 Single UNIX Specification:

- 48790 • The statement that *errno* is set on error is added.
48791 • The [EBADF] and [ENOTTY] optional error conditions are added.

48792 **NAME**

48793 twalk — traverse a binary search tree

48794 **SYNOPSIS**

48795 XSI #include <search.h>

48796 void twalk(const void *root,

48797 void (*action)(const void *, VISIT, int));

48798

48799 **DESCRIPTION**

48800 Refer to *tdelete()*.

48801 NAME

48802 *daylight*, *timezone*, *tzname*, *tzset* — set timezone conversion information

48803 SYNOPSIS

```
48804     #include <time.h>
48805 XSI     extern int daylight;
48806         extern long timezone;
48807 CX      extern char *tzname[2];
48808         void tzset(void);
48809
```

48810 DESCRIPTION

48811 The *tzset()* function shall use the value of the environment variable *TZ* to set time conversion
48812 information used by *ctime()*, *localtime()*, *mktme()*, and *strftime()*. If *TZ* is absent from the
48813 environment, implementation-defined default timezone information shall be used.

48814 The *tzset()* function shall set the external variable *tzname* as follows:

```
48815 tzname[0] = "std";
48816 tzname[1] = "dst";
```

48817 where *std* and *dst* are as described in the Base Definitions volume of IEEE Std 1003.1-2001,
48818 Chapter 8, Environment Variables.

48819 XSI The *tzset()* function also shall set the external variable *daylight* to 0 if Daylight Savings Time
48820 conversions should never be applied for the timezone in use; otherwise, non-zero. The external
48821 variable *timezone* shall be set to the difference, in seconds, between Coordinated Universal Time
48822 (UTC) and local standard time.

48823 RETURN VALUE

48824 The *tzset()* function shall not return a value.

48825 ERRORS

48826 No errors are defined.

48827 EXAMPLES

48828 Example *TZ* variables and their timezone differences are given in the table below:

<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

48837 APPLICATION USAGE

48838 None.

48839 RATIONALE

48840 None.

48841 FUTURE DIRECTIONS

48842 None.

48843 SEE ALSO

48844 *ctime()*, *localtime()*, *mktime()*, *strftime()*, the Base Definitions volume of IEEE Std 1003.1-2001,
48845 Chapter 8, Environment Variables, <time.h>

48846 CHANGE HISTORY

48847 First released in Issue 1. Derived from Issue 1 of the SVID.

48848 Issue 6

48849 The example is corrected.

48850 NAME

48851 *ualarm* — set the interval timer

48852 SYNOPSIS

48853 OB XSI #include <unistd.h>
48854 useconds_t ualarm(useconds_t *useconds*, useconds_t *interval*);
48855

48856 DESCRIPTION

48857 The *ualarm*() function shall cause the SIGALRM signal to be generated for the calling process
48858 after the number of realtime microseconds specified by the *useconds* argument has elapsed.
48859 When the *interval* argument is non-zero, repeated timeout notification occurs with a period in
48860 microseconds specified by the *interval* argument. If the notification signal, SIGALRM, is not
48861 caught or ignored, the calling process is terminated.

48862 Implementations may place limitations on the granularity of timer values. For each interval
48863 timer, if the requested timer value requires a finer granularity than the implementation supports,
48864 the actual timer value shall be rounded up to the next supported value.

48865 Interactions between *ualarm*() and any of the following are unspecified:

48866 *alarm*()
48867 *nanosleep*()
48868 *setitimer*()
48869 *timer_create*()
48870 *timer_delete*()
48871 *timer_getoverrun*()
48872 *timer_gettime*()
48873 *timer_settime*()
48874 *sleep*()

48875 RETURN VALUE

48876 The *ualarm*() function shall return the number of microseconds remaining from the previous
48877 *ualarm*() call. If no timeouts are pending or if *ualarm*() has not previously been called, *ualarm*()
48878 shall return 0.

48879 ERRORS

48880 No errors are defined.

48881 EXAMPLES

48882 None.

48883 APPLICATION USAGE

48884 Applications are recommended to use *nanosleep*() if the Timers option is supported, or
48885 *setitimer*(), *timer_create*(), *timer_delete*(), *timer_getoverrun*(), *timer_gettime*(), or *timer_settime*()
48886 instead of this function.

48887 RATIONALE

48888 None.

48889 FUTURE DIRECTIONS

48890 None.

48891 SEE ALSO

48892 *alarm*(), *nanosleep*(), *setitimer*(), *sleep*(), *timer_create*(), *timer_delete*(), *timer_getoverrun*(), the Base
48893 Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

48894 CHANGE HISTORY

48895 First released in Issue 4, Version 2.

48896 Issue 5

48897 Moved from X/OPEN UNIX extension to BASE.

48898 Issue 6

48899 This function is marked obsolescent.

48900 NAME

48901 *ulimit* — get and set process limits

48902 SYNOPSIS

48903 XSI `#include <ulimit.h>`

48904 `long ulimit(int cmd, ...);`

48905

48906 DESCRIPTION

48907 The *ulimit()* function shall control process limits. The process limits that can be controlled by
48908 this function include the maximum size of a single file that can be written (this is equivalent to
48909 using *setrlimit()* with RLIMIT_FSIZE). The *cmd* values, defined in <**ulimit.h**>, include:

48910 **UL_GETFSIZE** Return the file size limit (RLIMIT_FSIZE) of the process. The limit shall be in
48911 units of 512-byte blocks and shall be inherited by child processes. Files of any
48912 size can be read. The return value shall be the integer part of the soft file size
48913 limit divided by 512. If the result cannot be represented as a **long**, the result is
48914 unspecified.

48915 **UL_SETFSIZE** Set the file size limit for output operations of the process to the value of the
48916 second argument, taken as a **long**, multiplied by 512. If the result would
48917 overflow an **rlim_t**, the actual value set is unspecified. Any process may
48918 decrease its own limit, but only a process with appropriate privileges may
48919 increase the limit. The return value shall be the integer part of the new file size
48920 limit divided by 512.

48921 The *ulimit()* function shall not change the setting of *errno* if successful.

48922 As all return values are permissible in a successful situation, an application wishing to check for
48923 error situations should set *errno* to 0, then call *ulimit()*, and, if it returns -1, check to see if *errno* is
48924 non-zero.

48925 RETURN VALUE

48926 Upon successful completion, *ulimit()* shall return the value of the requested limit. Otherwise, -1
48927 shall be returned and *errno* set to indicate the error.

48928 ERRORS

48929 The *ulimit()* function shall fail and the limit shall be unchanged if:

48930 [EINVAL] The *cmd* argument is not valid.

48931 [EPERM] A process not having appropriate privileges attempts to increase its file size
48932 limit.

48933 EXAMPLES

48934 None.

48935 APPLICATION USAGE

48936 None.

48937 RATIONALE

48938 None.

48939 FUTURE DIRECTIONS

48940 None.

48941 **SEE ALSO**48942 *getrlimit()*, *setrlimit()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**ulimit.h**>48943 **CHANGE HISTORY**

48944 First released in Issue 1. Derived from Issue 1 of the SVID.

48945 **Issue 5**48946 In the description of **UL_SETFSIZE**, the text is corrected to refer to **rlim_t** rather than the
48947 spurious **rlimit_t**.48948 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

48949 NAME

48950 umask — set and get the file mode creation mask

48951 SYNOPSIS

```
48952 #include <sys/stat.h>
```

48953 mode_t umask(mode_t cmask);

48954 DESCRIPTION

48955 The `umask()` function shall set the process' file mode creation mask to *cmask* and return the
48956 previous value of the mask. Only the file permission bits of *cmask* (see `<sys/stat.h>`) are used; the
48957 meaning of the other bits is implementation-defined.

The process' file mode creation mask is used to turn off permission bits in the *mode* argument supplied during calls to the following functions:

- *open()*, *creat()*, *mkdir()*, and *mkdir()*
 - *mknod()*
 - *mq_open()*
 - *sem_open()*

48964 Bit positions that are set in *cmask* are cleared in the mode of the created file.

48965 RETURN VALUE

48966 The file permission bits in the value returned by *umask()* shall be the previous value of the file
48967 mode creation mask. The state of any other bits in that value is unspecified, except that a
48968 subsequent call to *umask()* with the returned value as *cmask* shall leave the state of the mask the
48969 same as its state before the first call, including any unspecified use of those bits.

48970 ERRORS

48971 No errors are defined.

48972 EXAMPLES

48973 **None.**

48974 APPLICATION

— 10 —

48976 **RATIONALE** Unsigned argument and return types for *umask()* were proposed. The return type and the argument types were both changed to *mode_t*.

48979 Historical implementations have made use of additional bits in *cmask* for their implementation-
48980 defined purposes. The addition of the text that the meaning of other bits of the field is
48981 implementation-defined permits these implementations to conform to this volume of
48982 IEEE Std 1003.1-2001.

48983 FUTURE DIRECTIONS

18984 None

48985 SEE ALSO

48986 *creat()*, *mkdir()*, *mknod()*, *mq_open()*, *open()*, *sem_open()*, the Base Definitions volume of 2
48987 IEEE Std 1003.1-2001, <sys/stat.h>, <sys/types.h>

48988 CHANGE HISTORY

48989 First released in Issue 1. Derived from Issue 1 of the SVID.

48990 Issue 6

48991 In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.

48992 The following new requirements on POSIX implementations derive from alignment with the
48993 Single UNIX Specification:

- 48994 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
48995 required for conforming implementations of previous POSIX specifications, it was not
48996 required for UNIX applications.

48997 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/143 is applied, adding the *mknod()*, 2
48998 *mq_open()*, and *sem_open()* functions to the DESCRIPTION and SEE ALSO sections. 2

48999 NAME

49000 **uname** — get the name of the current system

49001 SYNOPSIS

```
49002       #include <sys/utsname.h>
49003       int uname(struct utsname *name);
```

49004 DESCRIPTION

49005 The **uname()** function shall store information identifying the current system in the structure pointed to by *name*.

49007 The **uname()** function uses the **utsname** structure defined in <sys/utsname.h>.

49008 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.

49013 The format of each member is implementation-defined.

49014 RETURN VALUE

49015 Upon successful completion, a non-negative value shall be returned. Otherwise, -1 shall be returned and *errno* set to indicate the error.

49017 ERRORS

49018 No errors are defined.

49019 EXAMPLES

49020 None.

49021 APPLICATION USAGE

49022 The inclusion of the *nodename* member in this structure does not imply that it is sufficient information for interfacing to communications networks.

49024 RATIONALE

49025 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>.

49028 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.

49032 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.

49035 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.

49039 FUTURE DIRECTIONS

49040 None.

49041 SEE ALSO

49042 The Base Definitions volume of IEEE Std 1003.1-2001, <sys/utsname.h>

49043 CHANGE HISTORY

49044 First released in Issue 1. Derived from Issue 1 of the SVID.

49045 **NAME**

49046 *ungetc* — push byte back into input stream

49047 **SYNOPSIS**

49048 #include <stdio.h>
49049 int ungetc(int *c*, FILE **stream*);

49050 **DESCRIPTION**

49051 CX 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.

49054 The *ungetc()* function shall push the byte specified by *c* (converted to an **unsigned char**) back onto the input stream pointed to by *stream*. The pushed-back bytes shall be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file-positioning function (*fseek()*, *fsetpos()*, or *rewind()*) shall discard any pushed-back bytes for the stream. The external storage corresponding to the stream shall be unchanged.

49060 One byte of push-back shall be provided. If *ungetc()* is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail.

49062 If the value of *c* equals that of the macro EOF, the operation shall fail and the input stream shall be left unchanged.

49064 A successful call to *ungetc()* shall clear the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back bytes shall be the same as it was before the bytes were pushed back. The file-position indicator is decremented by each successful call to *ungetc()*; if its value was 0 before a call, its value is unspecified after the call.

49069 **RETURN VALUE**

49070 Upon successful completion, *ungetc()* shall return the byte pushed back after conversion.
49071 Otherwise, it shall return EOF.

49072 **ERRORS**

49073 No errors are defined.

49074 **EXAMPLES**

49075 None.

49076 **APPLICATION USAGE**

49077 None.

49078 **RATIONALE**

49079 None.

49080 **FUTURE DIRECTIONS**

49081 None.

49082 **SEE ALSO**

49083 *fseek()*, *getc()*, *fsetpos()*, *read()*, *rewind()*, *setbuf()*, the Base Definitions volume of
49084 IEEE Std 1003.1-2001, <stdio.h>

49085 **CHANGE HISTORY**

49086 First released in Issue 1. Derived from Issue 1 of the SVID.

49087 NAME

49088 ungetwc — push wide-character code back into the input stream

49089 SYNOPSIS

```
49090        #include <stdio.h>
49091        #include <wchar.h>
49092        wint_t ungetwc(wint_t wc, FILE *stream);
```

49093 DESCRIPTION

49094 CX 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.

49097 The *ungetwc()* function shall push the character corresponding to the wide-character code specified by *wc* back onto the input stream pointed to by *stream*. The pushed-back characters shall be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file-positioning function (*fseek()*, *fsetpos()*, or *rewind()*) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

49103 At least one character of push-back shall be provided. If *ungetwc()* is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail.

49106 If the value of *wc* equals that of the macro WEOF, the operation shall fail and the input stream shall be left unchanged.

49108 A successful call to *ungetwc()* shall clear the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back. The file-position indicator is decremented (by one or more) by each successful call to *ungetwc()*; if its value was 0 before a call, its value is unspecified after the call.

49113 RETURN VALUE

49114 Upon successful completion, *ungetwc()* shall return the wide-character code corresponding to the pushed-back character. Otherwise, it shall return WEOF.

49116 ERRORS

49117 The *ungetwc()* function may fail if:

49118 CX [EILSEQ] An invalid character sequence is detected, or a wide-character code does not correspond to a valid character.

49120 EXAMPLES

49121 None.

49122 APPLICATION USAGE

49123 None.

49124 RATIONALE

49125 None.

49126 FUTURE DIRECTIONS

49127 None.

49128 **SEE ALSO**

49129 *fseek()*, *fsetpos()*, *read()*, *rewind()*, *setbuf()*, the Base Definitions volume of IEEE Std 1003.1-2001,
49130 *<stdio.h>*, *<wchar.h>*

49131 **CHANGE HISTORY**

49132 First released in Issue 4. Derived from the MSE working draft.

49133 **Issue 5**

49134 The Optional Header (OH) marking is removed from *<stdio.h>*.

49135 **Issue 6**

49136 The [EILSEQ] optional error condition is marked CX.

49137 NAME

49138 unlink — remove a directory entry

49139 SYNOPSIS

```
49140 #include <unistd.h>
49141 int unlink(const char *path);
```

49142 DESCRIPTION

49143 The *unlink()* function shall remove a link to a file. If *path* names a symbolic link, *unlink()* shall
 49144 remove the symbolic link named by *path* and shall not affect any file or directory named by the
 49145 contents of the symbolic link. Otherwise, *unlink()* shall remove the link named by the pathname
 49146 pointed to by *path* and shall decrement the link count of the file referenced by the link.

49147 When the file's link count becomes 0 and no process has the file open, the space occupied by the
 49148 file shall be freed and the file shall no longer be accessible. If one or more processes have the file
 49149 open when the last link is removed, the link shall be removed before *unlink()* returns, but the
 49150 removal of the file contents shall be postponed until all references to the file are closed.

49151 The *path* argument shall not name a directory unless the process has appropriate privileges and
 49152 the implementation supports using *unlink()* on directories.

49153 Upon successful completion, *unlink()* shall mark for update the *st_ctime* and *st_mtime* fields of
 49154 the parent directory. Also, if the file's link count is not 0, the *st_ctime* field of the file shall be
 49155 marked for update.

49156 RETURN VALUE

49157 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* set to
 49158 indicate the error. If -1 is returned, the named file shall not be changed.

49159 ERRORS

49160 The *unlink()* function shall fail and shall not unlink the file if:

49161 [EACCES] Search permission is denied for a component of the path prefix, or write
 49162 permission is denied on the directory containing the directory entry to be
 49163 removed.

49164 [EBUSY] The file named by the *path* argument cannot be unlinked because it is being
 49165 used by the system or another process and the implementation considers this
 49166 an error.

49167 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
 49168 argument.

49169 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
 49170 component is longer than {NAME_MAX}.

49172 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

49173 [ENOTDIR] A component of the path prefix is not a directory.

49174 [EPERM] The file named by *path* is a directory, and either the calling process does not
 49175 have appropriate privileges, or the implementation prohibits using *unlink()*
 49176 on directories.

49177 XSI [EPERM] or [EACCES] The S_ISVTX flag is set on the directory containing the file referred to by the
 49178 *path* argument and the caller is not the file owner, nor is the caller the
 49179 directory owner, nor does the caller have appropriate privileges.

49181 [EROFS] The directory entry to be unlinked is part of a read-only file system.

49182 The *unlink()* function may fail and not unlink the file if:

49183 XSI [EBUSY] The file named by *path* is a named STREAM.

49184 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
49185 resolution of the *path* argument.

49186 [ENAMETOOLONG]
49187 As a result of encountering a symbolic link in resolution of the *path* argument,
49188 the length of the substituted pathname string exceeded {PATH_MAX}.

49189 [ETXTBSY] The entry to be unlinked is the last directory entry to a pure procedure (shared
49190 text) file that is being executed.

49191 EXAMPLES

49192 Removing a Link to a File

49193 The following example shows how to remove a link to a file named **/home/cnd/mod1** by
49194 removing the entry named **/modules/pass1**.

```
49195 #include <unistd.h>
49196 char *path = "/modules/pass1";
49197 int status;
49198 ...
49199 status = unlink(path);
```

49200 Checking for an Error

49201 The following example fragment creates a temporary password lock file named **LOCKFILE**,
49202 which is defined as **/etc/ptmp**, and gets a file descriptor for it. If the file cannot be opened for
49203 writing, *unlink()* is used to remove the link between the file descriptor and **LOCKFILE**.

```
49204 #include <sys/types.h>
49205 #include <stdio.h>
49206 #include <fcntl.h>
49207 #include <errno.h>
49208 #include <unistd.h>
49209 #include <sys/stat.h>
49210 #define LOCKFILE "/etc/ptmp"
49211 int pfd; /* Integer for file descriptor returned by open call. */
49212 FILE *fpfd; /* File pointer for use in putpwent(). */
49213 ...
49214 /* Open password Lock file. If it exists, this is an error. */
49215 if ((pfd = open(LOCKFILE, O_WRONLY| O_CREAT | O_EXCL, S_IRUSR
49216 | S_IWUSR | S_IRGRP | S_IROTH)) == -1) {
49217     fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
49218     exit(1);
49219 }
49220 /* Lock file created; proceed with fdopen of lock file so that
49221     putpwent() can be used.
49222 */
49223 if ((fpfd = fdopen(pfd, "w")) == NULL) {
```

```
49224     close(pfd);
49225     unlink(LOCKFILE);
49226     exit(1);
49227 }
```

49228 Replacing Files

49229 The following example fragment uses *unlink()* to discard links to files, so that they can be
49230 replaced with new versions of the files. The first call removes the link to **LOCKFILE** if an error
49231 occurs. Successive calls remove the links to **SAVEFILE** and **PASSWDFILE** so that new links can
49232 be created, then removes the link to **LOCKFILE** when it is no longer needed.

```
49233 #include <sys/types.h>
49234 #include <stdio.h>
49235 #include <fcntl.h>
49236 #include <errno.h>
49237 #include <unistd.h>
49238 #include <sys/stat.h>

49239 #define LOCKFILE "/etc/ptmp"
49240 #define PASSWDFILE "/etc/passwd"
49241 #define SAVEFILE "/etc/opasswd"
49242 ...
49243 /* If no change was made, assume error and leave passwd unchanged. */
49244 if (!valid_change) {
49245     fprintf(stderr, "Could not change password for user %s\n", user);
49246     unlink(LOCKFILE);
49247     exit(1);
49248 }
49249 /* Change permissions on new password file. */
49250 chmod(LOCKFILE, S_IRUSR | S_IRGRP | S_IROTH);

49251 /* Remove saved password file. */
49252 unlink(SAVEFILE);

49253 /* Save current password file. */
49254 link(PASSWDFILE, SAVEFILE);

49255 /* Remove current password file. */
49256 unlink(PASSWDFILE);

49257 /* Save new password file as current password file. */
49258 link(LOCKFILE, PASSWDFILE);

49259 /* Remove lock file. */
49260 unlink(LOCKFILE);

49261 exit(0);
```

49262 APPLICATION USAGE

49263 Applications should use *rmdir()* to remove a directory.

49264 RATIONALE

49265 Unlinking a directory is restricted to the superuser in many historical implementations for
49266 reasons given in *link()* (see also *rename()*).

49267 The meaning of [EBUSY] in historical implementations is “mount point busy”. Since this volume
49268 of IEEE Std 1003.1-2001 does not cover the system administration concepts of mounting and
49269 unmounting, the description of the error was changed to “resource busy”. (This meaning is used
49270 by some device drivers when a second process tries to open an exclusive use device.) The
49271 wording is also intended to allow implementations to refuse to remove a directory if it is the
49272 root or current working directory of any process.

49273 FUTURE DIRECTIONS

49274 None.

49275 SEE ALSO

49276 *close()*, *link()*, *remove()*, *rmdir()*, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

49277 CHANGE HISTORY

49278 First released in Issue 1. Derived from Issue 1 of the SVID.

49279 Issue 5

49280 The [EBUSY] error is added to the optional part of the ERRORS section.

49281 Issue 6

49282 The following new requirements on POSIX implementations derive from alignment with the
49283 Single UNIX Specification:

- 49284 • In the DESCRIPTION, the effect is specified if *path* specifies a symbolic link.
- 49285 • The [ELOOP] mandatory error condition is added.
- 49286 • A second [ENAMETOOLONG] is added as an optional error condition.
- 49287 • The [ETXTBSY] optional error condition is added.

49288 The following changes were made to align with the IEEE P1003.1a draft standard:

- 49289 • The [ELOOP] optional error condition is added.

49290 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

49291 NAME

49292 unlockpt — unlock a pseudo-terminal master/slave pair

49293 SYNOPSIS

49294 XSI #include <stdlib.h>

49295 int unlockpt(int *fildes*);

49296

49297 DESCRIPTION

49298 The *unlockpt()* function shall unlock the slave pseudo-terminal device associated with the
49299 master to which *fildes* refers.

49300 Conforming applications shall ensure that they call *unlockpt()* before opening the slave side of a
49301 pseudo-terminal device.

49302 RETURN VALUE

49303 Upon successful completion, *unlockpt()* shall return 0. Otherwise, it shall return -1 and set *errno*
49304 to indicate the error.

49305 ERRORS

49306 The *unlockpt()* function may fail if:

49307 [EBADF] The *fildes* argument is not a file descriptor open for writing.

49308 [EINVAL] The *fildes* argument is not associated with a master pseudo-terminal device.

49309 EXAMPLES

49310 None.

49311 APPLICATION USAGE

49312 None.

49313 RATIONALE

49314 None.

49315 FUTURE DIRECTIONS

49316 None.

49317 SEE ALSO

49318 *grantpt()*, *open()*, *ptsname()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

49319 CHANGE HISTORY

49320 First released in Issue 4, Version 2.

49321 Issue 5

49322 Moved from X/OPEN UNIX extension to BASE.

49323 Issue 6

49324 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

49325 **NAME**

49326 unsetenv — remove an environment variable

49327 **SYNOPSIS**

49328 CX #include <stdlib.h>

49329 int unsetenv(const char *name);

49330

49331 **DESCRIPTION**

49332 The *unsetenv()* function shall remove an environment variable from the environment of the calling process. The *name* argument points to a string, which is the name of the variable to be removed. The named argument shall not contain an '=' character. If the named variable does not exist in the current environment, the environment shall be unchanged and the function is considered to have completed successfully.

49337 If the application modifies *environ* or the pointers to which it points, the behavior of *unsetenv()* is undefined. The *unsetenv()* function shall update the list of pointers to which *environ* points.

49339 The *unsetenv()* function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

49341 **RETURN VALUE**

49342 Upon successful completion, zero shall be returned. Otherwise, -1 shall be returned, *errno* set to indicate the error, and the environment shall be unchanged.

49344 **ERRORS**

49345 The *unsetenv()* function shall fail if:

49346 [EINVAL] The *name* argument is a null pointer, points to an empty string, or points to a string containing an '=' character.

49348 **EXAMPLES**

49349 None.

49350 **APPLICATION USAGE**

49351 None.

49352 **RATIONALE**

49353 Refer to the RATIONALE section in *setenv()*.

49354 **FUTURE DIRECTIONS**

49355 None.

49356 **SEE ALSO**

49357 *getenv()*, *setenv()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<stdlib.h>**,
49358 **<sys/types.h>**, **<unistd.h>**

49359 **CHANGE HISTORY**

49360 First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

49361 NAME

49362 usleep — suspend execution for an interval

49363 SYNOPSIS

49364 OB XSI #include <unistd.h>

49365 int usleep(useconds_t useconds);

49366

49367 DESCRIPTION

49368 The *usleep()* function shall cause the calling thread to be suspended from execution until either
 49369 the number of realtime microseconds specified by the argument *useconds* has elapsed or a signal
 49370 is delivered to the calling thread and its action is to invoke a signal-catching function or to
 49371 terminate the process. The suspension time may be longer than requested due to the scheduling
 49372 of other activity by the system.

49373 The *useconds* argument shall be less than one million. If the value of *useconds* is 0, then the call
 49374 has no effect.

49375 If a SIGALRM signal is generated for the calling process during execution of *usleep()* and if the
 49376 SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *usleep()*
 49377 returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also
 49378 unspecified whether it remains pending after *usleep()* returns or it is discarded.

49379 If a SIGALRM signal is generated for the calling process during execution of *usleep()*, except as a
 49380 result of a prior call to *alarm()*, and if the SIGALRM signal is not being ignored or blocked from
 49381 delivery, it is unspecified whether that signal has any effect other than causing *usleep()* to return.

49382 If a signal-catching function interrupts *usleep()* and examines or changes either the time a
 49383 SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or
 49384 whether the SIGALRM signal is blocked from delivery, the results are unspecified.

49385 If a signal-catching function interrupts *usleep()* and calls *siglongjmp()* or *longjmp()* to restore an
 49386 environment saved prior to the *usleep()* call, the action associated with the SIGALRM signal and
 49387 the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also
 49388 unspecified whether the SIGALRM signal is blocked, unless the thread's signal mask is restored
 49389 as part of the environment. 2 2

49390 Implementations may place limitations on the granularity of timer values. For each interval
 49391 timer, if the requested timer value requires a finer granularity than the implementation supports,
 49392 the actual timer value shall be rounded up to the next supported value.

49393 Interactions between *usleep()* and any of the following are unspecified:

49394 *nanosleep()*
 49395 *setitimer()*
 49396 *timer_create()*
 49397 *timer_delete()*
 49398 *timer_getoverrun()*
 49399 *timer_gettime()*
 49400 *timer_settime()*
 49401 *ualarm()*
 49402 *sleep()*

49403 The *usleep()* function need not be reentrant. A function that is not required to be reentrant is not 2 2
 49404 required to be thread-safe.

49405 RETURN VALUE

49406 Upon successful completion, *usleep()* shall return 0; otherwise, it shall return -1 and set *errno* to indicate the error.

49408 ERRORS

49409 The *usleep()* function may fail if:

49410 [EINVAL] The time interval specified one million or more microseconds.

49411 EXAMPLES

49412 None.

49413 APPLICATION USAGE

49414 Applications are recommended to use *nanosleep()* if the Timers option is supported, or
49415 *settimer()*, *timer_create()*, *timer_delete()*, *timer_getoverrun()*, *timer_gettime()*, or *timer_settime()*
49416 instead of this function.

49417 RATIONALE

49418 None.

49419 FUTURE DIRECTIONS

49420 None.

49421 SEE ALSO

49422 *alarm()*, *getitimer()*, *nanosleep()*, *sigaction()*, *sleep()*, *timer_create()*, *timer_delete()*,
49423 *timer_getoverrun()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**unistd.h**>

49424 CHANGE HISTORY

49425 First released in Issue 4, Version 2.

49426 Issue 5

49427 Moved from X/OPEN UNIX extension to BASE.

49428 The DESCRIPTION is changed to indicate that timers are now thread-based rather than
49429 process-based.

49430 Issue 6

49431 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

49432 This function is marked obsolescent.

49433 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/144 is applied, updating the 2
49434 DESCRIPTION from “process’ signal mask” to “thread’s signal mask”, and adding a statement 2
49435 that the *usleep()* function need not be reentrant. 2

49436 NAME

49437 utime — set file access and modification times

49438 SYNOPSIS

49439 #include <utime.h>
49440 int utime(const char *path, const struct utimbuf *times);

49441 DESCRIPTION

49442 The *utime()* function shall set the access and modification times of the file named by the *path* argument.

49444 If *times* is a null pointer, the access and modification times of the file shall be set to the current time. The effective user ID of the process shall match the owner of the file, or the process has write permission to the file or has appropriate privileges, to use *utime()* in this manner.

49447 If *times* is not a null pointer, *times* shall be interpreted as a pointer to a **utimbuf** structure and the access and modification times shall be set to the values contained in the designated structure. Only a process with the effective user ID equal to the user ID of the file or a process with appropriate privileges may use *utime()* this way.

49451 The **utimbuf** structure is defined in the <utime.h> header. The times in the structure **utimbuf** are measured in seconds since the Epoch.

49453 Upon successful completion, *utime()* shall mark the time of the last file status change, *st_ctime*, to be updated; see <sys/stat.h>.

49455 RETURN VALUE

49456 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* shall
49457 be set to indicate the error, and the file times shall not be affected.

49458 ERRORS

49459 The *utime()* function shall fail if:

49460 [EACCES] Search permission is denied by a component of the path prefix; or the *times* argument is a null pointer and the effective user ID of the process does not match the owner of the file, the process does not have write permission for the file, and the process does not have appropriate privileges.

49464 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

49466 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

49469 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

49470 [ENOTDIR] A component of the path prefix is not a directory.

49471 [EPERM] The *times* argument is not a null pointer and the calling process' effective user ID does not match the owner of the file and the calling process does not have the appropriate privileges.

49474 [EROFS] The file system containing the file is read-only.

49475 The *utime()* function may fail if:

49476 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

49478 [ENAMETOOLONG]

49479 As a result of encountering a symbolic link in resolution of the *path* argument,
49480 the length of the substituted pathname string exceeded {PATH_MAX}.

49481 **EXAMPLES**

49482 None.

49483 **APPLICATION USAGE**

49484 None.

49485 **RATIONALE**

49486 The *actime* structure member must be present so that an application may set it, even though an
49487 implementation may ignore it and not change the access time on the file. If an application
49488 intends to leave one of the times of a file unchanged while changing the other, it should use
49489 *stat()* to retrieve the file's *st_atime* and *st_mtime* parameters, set *actime* and *modtime* in the buffer,
49490 and change one of them before making the *utime()* call.

49491 **FUTURE DIRECTIONS**

49492 None.

49493 **SEE ALSO**

49494 The Base Definitions volume of IEEE Std 1003.1-2001, <sys/stat.h>, <utime.h>

49495 **CHANGE HISTORY**

49496 First released in Issue 1. Derived from Issue 1 of the SVID.

49497 **Issue 6**

49498 The following new requirements on POSIX implementations derive from alignment with the
49499 Single UNIX Specification:

- 49500 • The requirement to include <sys/types.h> has been removed. Although <sys/types.h> was
49501 required for conforming implementations of previous POSIX specifications, it was not
49502 required for UNIX applications.
- 49503 • The [ELOOP] mandatory error condition is added.
- 49504 • A second [ENAMETOOLONG] is added as an optional error condition.

49505 The following changes were made to align with the IEEE P1003.1a draft standard:

- 49506 • The [ELOOP] optional error condition is added.

49507 The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

49508 NAME

49509 utimes — set file access and modification times (LEGACY)

49510 SYNOPSIS

49511 XSI #include <sys/time.h>

49512 int utimes(const char *path, const struct timeval times[2]);

49513

49514 DESCRIPTION

49515 The *utimes()* function shall set the access and modification times of the file pointed to by the *path* argument to the value of the *times* argument. The *utimes()* function allows time specifications accurate to the microsecond.

49518 For *utimes()*, the *times* argument is an array of **timeval** structures. The first array member
49519 represents the date and time of last access, and the second member represents the date and time
49520 of last modification. The times in the **timeval** structure are measured in seconds and
49521 microseconds since the Epoch, although rounding toward the nearest second may occur.

49522 If the *times* argument is a null pointer, the access and modification times of the file shall be set to
49523 the current time. The effective user ID of the process shall match the owner of the file, or has
49524 write access to the file or appropriate privileges to use this call in this manner. Upon completion,
49525 *utimes()* shall mark the time of the last file status change, *st_ctime*, for update.

49526 RETURN VALUE

49527 Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and *errno* shall
49528 be set to indicate the error, and the file times shall not be affected.

49529 ERRORS

49530 The *utimes()* function shall fail if:

49531 [EACCES] Search permission is denied by a component of the path prefix; or the *times*
49532 argument is a null pointer and the effective user ID of the process does not
49533 match the owner of the file and write access is denied.

49534 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path*
49535 argument.

49536 [ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname
49537 component is longer than {NAME_MAX}.

49539 [ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

49540 [ENOTDIR] A component of the path prefix is not a directory.

49541 [EPERM] The *times* argument is not a null pointer and the calling process' effective user
49542 ID has write access to the file but does not match the owner of the file and the
49543 calling process does not have the appropriate privileges.

49544 [EROFS] The file system containing the file is read-only.

49545 The *utimes()* function may fail if:

49546 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
49547 resolution of the *path* argument.

49548 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
49549 whose length exceeds {PATH_MAX}.

49551 EXAMPLES

49552 None.

49553 APPLICATION USAGE

49554 For applications portability, the *utime()* function should be used to set file access and
49555 modification times instead of *utimes()*.

49556 RATIONALE

49557 None.

49558 FUTURE DIRECTIONS

49559 This function may be withdrawn in a future version.

49560 SEE ALSO

49561 *utime()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/time.h>

49562 CHANGE HISTORY

49563 First released in Issue 4, Version 2.

49564 Issue 5

49565 Moved from X/OPEN UNIX extension to BASE.

49566 Issue 6

49567 This function is marked LEGACY.

49568 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

49569 The wording of the mandatory [ELOOP] error condition is updated, and a second optional
49570 [ELOOP] error condition is added.

49571 NAME

49572 va_arg, va_copy, va_end, va_start — handle variable argument list

49573 SYNOPSIS

```
49574     #include <stdarg.h>
49575     type va_arg(va_list ap, type);
49576     void va_copy(va_list dest, va_list src);
49577     void va_end(va_list ap);
49578     void va_start(va_list ap, argN);
```

49579 DESCRIPTION

49580 Refer to the Base Definitions volume of IEEE Std 1003.1-2001, <stdarg.h>.

49581 NAME

49582 vfork — create a new process; share virtual memory

49583 SYNOPSIS

49584 OB XSI #include <unistd.h>
49585 pid_t vfork(void);
49586

49587 DESCRIPTION

49588 The *vfork()* function shall be equivalent to *fork()*, except that the behavior is undefined if the
49589 process created by *vfork()* either modifies any data other than a variable of type **pid_t** used to
49590 store the return value from *vfork()*, or returns from the function in which *vfork()* was called, or
49591 calls any other function before successfully calling *_exit()* or one of the *exec* family of functions.

49592 RETURN VALUE

49593 Upon successful completion, *vfork()* shall return 0 to the child process and return the process ID
49594 of the child process to the parent process. Otherwise, -1 shall be returned to the parent, no child
49595 process shall be created, and *errno* shall be set to indicate the error.

49596 ERRORS

49597 The *vfork()* function shall fail if:

49598 [EAGAIN] The system-wide limit on the total number of processes under execution
49599 would be exceeded, or the system-imposed limit on the total number of
49600 processes under execution by a single user would be exceeded.
49601 [ENOMEM] There is insufficient swap space for the new process.

49602 EXAMPLES

49603 None.

49604 APPLICATION USAGE

49605 Conforming applications are recommended not to depend on *vfork()*, but to use *fork()* instead.
49606 The *vfork()* function may be withdrawn in a future version.

49607 On some implementations, *vfork()* is equivalent to *fork()*.

49608 The *vfork()* function differs from *fork()* only in that the child process can share code and data
49609 with the calling process (parent process). This speeds cloning activity significantly at a risk to
49610 the integrity of the parent process if *vfork()* is misused.

49611 The use of *vfork()* for any purpose except as a prelude to an immediate call to a function from
49612 the *exec* family, or to *_exit()*, is not advised.

49613 The *vfork()* function can be used to create new processes without fully copying the address
49614 space of the old process. If a forked process is simply going to call *exec*, the data space copied
49615 from the parent to the child by *fork()* is not used. This is particularly inefficient in a paged
49616 environment, making *vfork()* particularly useful. Depending upon the size of the parent's data
49617 space, *vfork()* can give a significant performance improvement over *fork()*.

49618 The *vfork()* function can normally be used just like *fork()*. It does not work, however, to return
49619 while running in the child's context from the caller of *vfork()* since the eventual return from
49620 *vfork()* would then return to a no longer existent stack frame. Care should be taken, also, to call
49621 *_exit()* rather than *exit()* if *exec* cannot be used, since *exit()* flushes and closes standard I/O
49622 channels, thereby damaging the parent process' standard I/O data structures. (Even with *fork()*,
49623 it is wrong to call *exit()*, since buffered data would then be flushed twice.)

49624 If signal handlers are invoked in the child process after *vfork()*, they must follow the same rules
49625 as other code in the child process.

49626 RATIONALE

49627 None.

49628 FUTURE DIRECTIONS

49629 This function may be withdrawn in a future version.

49630 SEE ALSO

49631 *exec*, *exit*(), *fork*(), *wait*(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

49632 CHANGE HISTORY

49633 First released in Issue 4, Version 2.

49634 Issue 5

49635 Moved from X/OPEN UNIX extension to BASE.

49636 Issue 6

49637 This function is marked obsolescent.

49638 NAME

49639 vfprintf, vprintf, vsnprintf, vsprintf — format output of a stdarg argument list

49640 SYNOPSIS

```
49641        #include <stdarg.h>
49642        #include <stdio.h>
49643        int vfprintf(FILE *restrict stream, const char *restrict format,
49644                    va_list ap);
49645        int vprintf(const char *restrict format, va_list ap);
49646        int vsnprintf(char *restrict s, size_t n, const char *restrict format,
49647                    va_list ap);
49648        int vsprintf(char *restrict s, const char *restrict format, va_list ap);
```

49649 DESCRIPTION

49650 CX The functionality described on this reference page is aligned with the ISO C standard. Any
49651 conflict between the requirements described here and the ISO C standard is unintentional. This
49652 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

49653 The *vprintf()*, *vfprintf()*, *vsnprintf()*, and *vsprintf()* functions shall be equivalent to *printf()*,
49654 *fprintf()*, *snprintf()*, and *sprintf()* respectively, except that instead of being called with a variable
49655 number of arguments, they are called with an argument list as defined by *<stdarg.h>*.

49656 These functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro,
49657 the value of *ap* after the return is unspecified.

49658 RETURN VALUE

49659 Refer to *fprintf()*.

49660 ERRORS

49661 Refer to *fprintf()*.

49662 EXAMPLES

49663 None.

49664 APPLICATION USAGE

49665 Applications using these functions should call *va_end(ap)* afterwards to clean up.

49666 RATIONALE

49667 None.

49668 FUTURE DIRECTIONS

49669 None.

49670 SEE ALSO

49671 *fprintf()*, the Base Definitions volume of IEEE Std 1003.1-2001, *<stdarg.h>*, *<stdio.h>*

49672 CHANGE HISTORY

49673 First released in Issue 1. Derived from Issue 1 of the SVID.

49674 Issue 5

49675 The *vsnprintf()* function is added.

49676 Issue 6

49677 The *vfprintf()*, *vprintf()*, *vsnprintf()*, and *vsprintf()* functions are updated for alignment with the
49678 ISO/IEC 9899: 1999 standard.

49679 NAME

49680 vfscanf, vscanf, vsscanf — format input of a stdarg argument list

49681 SYNOPSIS

```
49682        #include <stdarg.h>
49683        #include <stdio.h>
49684        int vfscanf(FILE *restrict stream, const char *restrict format,
49685                    va_list arg);
49686        int vscanf(const char *restrict format, va_list arg);
49687        int vsscanf(const char *restrict s, const char *restrict format,
49688                    va_list arg);
```

49689 DESCRIPTION

49690 CX 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.

49693 The *vscanf()*, *vfscanf()*, and *vsscanf()* functions shall be equivalent to the *scanf()*, *fscanf()*, and *sscanf()* functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the `<stdarg.h>` header. These functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro, the value of *ap* after the return is unspecified.

49698 RETURN VALUE

49699 Refer to *fscanf()*.

49700 ERRORS

49701 Refer to *fscanf()*.

49702 EXAMPLES

49703 None.

49704 APPLICATION USAGE

49705 Applications using these functions should call *va_end(ap)* afterwards to clean up.

49706 RATIONALE

49707 None.

49708 FUTURE DIRECTIONS

49709 None.

49710 SEE ALSO

49711 *fscanf()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdarg.h>`, `<stdio.h>`

49712 CHANGE HISTORY

49713 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

49714 NAME

49715 *vfwprintf*, *vswprintf*, *vwprintf* — wide-character formatted output of a stdarg argument list

49716 SYNOPSIS

```
49717        #include <stdarg.h>
49718        #include <stdio.h>
49719        #include <wchar.h>
49720
49720        int vfwprintf(FILE *restrict stream, const wchar_t *restrict format,
49721                va_list arg);
49722        int vswprintf(wchar_t *restrict ws, size_t n,
49723                const wchar_t *restrict format, va_list arg);
49724        int vwprintf(const wchar_t *restrict format, va_list arg);
```

49725 DESCRIPTION

49726 CX 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.

49729 The *vfwprintf()*, *vswprintf()*, and *vwprintf()* functions shall be equivalent to *fwprintf()*, *swprintf()*, and *wprintf()* respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `<stdarg.h>`.

49732 These functions shall not invoke the *va_end* macro. However, as these functions do invoke the *va_arg* macro, the value of *ap* after the return is unspecified.

49734 RETURN VALUE

49735 Refer to *fwprintf()*.

49736 ERRORS

49737 Refer to *fwprintf()*.

49738 EXAMPLES

49739 None.

49740 APPLICATION USAGE

49741 Applications using these functions should call *va_end(ap)* afterwards to clean up.

49742 RATIONALE

49743 None.

49744 FUTURE DIRECTIONS

49745 None.

49746 SEE ALSO

49747 *fwprintf()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdarg.h>`, `<stdio.h>`, `<wchar.h>`

49749 CHANGE HISTORY

49750 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
49751 (E).

49752 Issue 6

49753 The *vfwprintf()*, *vswprintf()*, and *vwprintf()* prototypes are updated for alignment with the
49754 ISO/IEC 9899:1999 standard. ()

49755 NAME

49756 vfwscanf, vswscanf, vwscanf — wide-character formatted input of a stdarg argument list

49757 SYNOPSIS

```
49758        #include <stdarg.h>
49759        #include <stdio.h>
49760        #include <wchar.h>
49761
49762        int vfwscanf(FILE *restrict stream, const wchar_t *restrict format,
49763                    va_list arg);
49764        int vswscanf(const wchar_t *restrict ws, const wchar_t *restrict format,
49765                    va_list arg);
49766        int vwscanf(const wchar_t *restrict format, va_list arg);
```

49766 DESCRIPTION

49767 CX 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.

49770 The *vfwscanf()*, *vswscanf()*, and *vwscanf()* functions shall be equivalent to the *fscanf()*, *swscanf()*, and *wscanf()* functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the `<stdarg.h>` header. These functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro, the value of *ap* after the return is unspecified.

49775 RETURN VALUE

49776 Refer to *fscanf()*.

49777 ERRORS

49778 Refer to *fscanf()*.

49779 EXAMPLES

49780 None.

49781 APPLICATION USAGE

49782 Applications using these functions should call *va_end(ap)* afterwards to clean up.

49783 RATIONALE

49784 None.

49785 FUTURE DIRECTIONS

49786 None.

49787 SEE ALSO

49788 *fscanf()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdarg.h>`, `<stdio.h>`,
49789 `<wchar.h>`

49790 CHANGE HISTORY

49791 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

49792 NAME

49793 `vprintf` — format the output of a stdarg argument list

49794 SYNOPSIS

```
49795        #include <stdarg.h>
49796        #include <stdio.h>
```

```
49797        int vprintf(const char *restrict format, va_list ap);
```

49798 DESCRIPTION

49799 Refer to `vfprintf()`.

49800 NAME

49801 `vscanf` — format input of a stdarg argument list

49802 SYNOPSIS

49803 `#include <stdarg.h>`

49804 `#include <stdio.h>`

49805 `int vscanf(const char *restrict format, va_list arg);`

49806 DESCRIPTION

49807 Refer to `vfscanf()`.

49808 NAME

49809 `vsnprintf, vsprintf — format output of a stdarg argument list`

49810 SYNOPSIS

```
49811        #include <stdarg.h>
49812        #include <stdio.h>
49813        int vsnprintf(char *restrict s, size_t n,
49814                 const char *restrict format, va_list ap);
49815        int vsprintf(char *restrict s, const char *restrict format,
49816                 va_list ap);
```

49817 DESCRIPTION

49818 Refer to `vfprintf()`.

49819 NAME

49820 vsscanf — format input of a stdarg argument list

49821 SYNOPSIS

```
49822        #include <stdarg.h>
49823        #include <stdio.h>
```

```
49824        int vsscanf(const char *restrict s, const char *restrict format,
49825                va_list arg);
```

49826 DESCRIPTION

49827 Refer to *vfscanf()*.

49828 NAME

49829 **vswprintf** — wide-character formatted output of a stdarg argument list

49830 SYNOPSIS

```
49831        #include <stdarg.h>
49832        #include <stdio.h>
49833        #include <wchar.h>
49834        int vswprintf(wchar_t *restrict ws, size_t n,
49835                    const wchar_t *restrict format, va_list arg);
```

49836 DESCRIPTION

49837 Refer to *vfwprintf()*.

49838 NAME

49839 vswscanf — wide-character formatted input of a stdarg argument list

49840 SYNOPSIS

```
49841        #include <stdarg.h>
49842        #include <stdio.h>
49843        #include <wchar.h>
49844        int vswscanf(const wchar_t *restrict ws, const wchar_t *restrict format,
49845                    va_list arg);
```

49846 DESCRIPTION

49847 Refer to *vfwscanf()*.

49848 NAME

49849 **vwprintf** — wide-character formatted output of a stdarg argument list

49850 SYNOPSIS

```
49851        #include <stdarg.h>
49852        #include <stdio.h>
49853        #include <wchar.h>
49854        int vwprintf(const wchar_t *restrict format, va_list arg);
```

49855 DESCRIPTION

49856 Refer to *vfwprintf()*.

49857 NAME

49858 vwscanf — wide-character formatted input of a stdarg argument list

49859 SYNOPSIS

```
49860       #include <stdarg.h>
49861       #include <stdio.h>
49862       #include <wchar.h>
49863       int vwscanf(const wchar_t *restrict format, va_list arg);
```

49864 DESCRIPTION

49865 Refer to *vfwscanf()*.

49866 NAME

49867 wait, waitpid — wait for a child process to stop or terminate

49868 SYNOPSIS

```
49869 #include <sys/wait.h>
49870 pid_t wait(int *stat_loc);
49871 pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

49872 DESCRIPTION

49873 The *wait()* and *waitpid()* functions shall obtain status information pertaining to one of the caller's child processes. Various options permit status information to be obtained for child processes that have terminated or stopped. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

49877 The *wait()* function shall suspend execution of the calling thread until status information for one of the terminated child processes of the calling process is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If more than one thread is suspended in *wait()* or *waitpid()* awaiting termination of the same process, exactly one thread shall return the process status at the time of the target process termination. If status information is available prior to the call to *wait()*, return shall be immediate.

49883 The *waitpid()* function shall be equivalent to *wait()* if the *pid* argument is **(pid_t)**-1 and the *options* argument is 0. Otherwise, its behavior shall be modified by the values of the *pid* and *options* arguments.

49886 The *pid* argument specifies a set of child processes for which *status* is requested. The *waitpid()* function shall only return the status of a child process from this set:

- 49888 • If *pid* is equal to **(pid_t)**-1, *status* is requested for any child process. In this respect, *waitpid()* is then equivalent to *wait()*.
- 49890 • If *pid* is greater than 0, it specifies the process ID of a single child process for which *status* is requested.
- 49892 • If *pid* is 0, *status* is requested for any child process whose process group ID is equal to that of the calling process.
- 49894 • If *pid* is less than **(pid_t)**-1, *status* is requested for any child process whose process group ID is equal to the absolute value of *pid*.

49896 The *options* argument is constructed from the bitwise-inclusive OR of zero or more of the following flags, defined in the **<sys/wait.h>** header:

49898 XSI	WCONTINUED	The <i>waitpid()</i> function shall report the status of any continued child process specified by <i>pid</i> whose status has not been reported since it continued from a job control stop.
-----------	-------------------	---

49901	WNOHANG	The <i>waitpid()</i> function shall not suspend execution of the calling thread if <i>status</i> is not immediately available for one of the child processes specified by <i>pid</i> .
-------	----------------	--

49904	WUNTRACED	The status of any child processes specified by <i>pid</i> that are stopped, and whose status has not yet been reported since they stopped, shall also be reported to the requesting process.
-------	------------------	--

49907 XSI	If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN , and the process has no unwaited-for children that were transformed into zombie processes, the calling thread shall block until all of the children of the process containing the calling thread terminate, and <i>wait()</i> and <i>waitpid()</i> shall fail and set <i>errno</i> to [ECHILD] .
-----------	---

49911 If *wait()* or *waitpid()* return because the status of a child process is available, these functions
 49912 shall return a value equal to the process ID of the child process. In this case, if the value of the
 49913 argument *stat_loc* is not a null pointer, information shall be stored in the location pointed to by
 49914 *stat_loc*. The value stored at the location pointed to by *stat_loc* shall be 0 if and only if the status
 49915 returned is from a terminated child process that terminated by one of the following means:

- 49916 1. The process returned 0 from *main()*.
- 49917 2. The process called *_exit()* or *exit()* with a *status* argument of 0.
- 49918 3. The process was terminated because the last thread in the process terminated.

49919 Regardless of its value, this information may be interpreted using the following macros, which
 49920 are defined in <sys/wait.h> and evaluate to integral expressions; the *stat_val* argument is the
 49921 integer value pointed to by *stat_loc*.

49922 **WIFEXITED(*stat_val*)**

49923 Evaluates to a non-zero value if *status* was returned for a child process that terminated
 49924 normally.

49925 **WEXITSTATUS(*stat_val*)**

49926 If the value of **WIFEXITED(*stat_val*)** is non-zero, this macro evaluates to the low-order 8 bits
 49927 of the *status* argument that the child process passed to *_exit()* or *exit()*, or the value the child
 49928 process returned from *main()*.

49929 **WIFSIGNALED(*stat_val*)**

49930 Evaluates to a non-zero value if *status* was returned for a child process that terminated due
 49931 to the receipt of a signal that was not caught (see <signal.h>).

49932 **WTERMSIG(*stat_val*)**

49933 If the value of **WIFSIGNALED(*stat_val*)** is non-zero, this macro evaluates to the number of
 49934 the signal that caused the termination of the child process.

49935 **WIFSTOPPED(*stat_val*)**

49936 Evaluates to a non-zero value if *status* was returned for a child process that is currently
 49937 stopped.

49938 **WSTOPSIG(*stat_val*)**

49939 If the value of **WIFSTOPPED(*stat_val*)** is non-zero, this macro evaluates to the number of the
 49940 signal that caused the child process to stop.

49941 XSI **WIFCONTINUED(*stat_val*)**

49942 Evaluates to a non-zero value if *status* was returned for a child process that has continued
 49943 from a job control stop.

49944 SPN It is unspecified whether the *status* value returned by calls to *wait()* or *waitpid()* for processes
 49945 created by *posix_spawn()* or *posix_spawnp()* can indicate a **WIFSTOPPED(*stat_val*)** before
 49946 subsequent calls to *wait()* or *waitpid()* indicate **WIFEXITED(*stat_val*)** as the result of an error
 49947 detected before the new process image starts executing.

49948 It is unspecified whether the *status* value returned by calls to *wait()* or *waitpid()* for processes
 49949 created by *posix_spawn()* or *posix_spawnp()* can indicate a **WIFSIGNALED(*stat_val*)** if a signal is
 49950 sent to the parent's process group after *posix_spawn()* or *posix_spawnp()* is called.

49951 If the information pointed to by *stat_loc* was stored by a call to *waitpid()* that specified the
 49952 XSI **WUNTRACED** flag and did not specify the **WCONTINUED** flag, exactly one of the macros
 49953 **WIFEXITED(**stat_loc*)**, **WIFSIGNALED(**stat_loc*)**, and **WIFSTOPPED(**stat_loc*)** shall evaluate to
 49954 a non-zero value.

49955 If the information pointed to by *stat_loc* was stored by a call to *waitpid()* that specified the
49956 XSI WUNTRACED and WCONTINUED flags, exactly one of the macros WIFEXITED(**stat_loc*),
49957 XSI WIFSIGNALLED(**stat_loc*), WIFSTOPPED(**stat_loc*), and WIFCONTINUED(**stat_loc*) shall
49958 evaluate to a non-zero value.

49959 If the information pointed to by *stat_loc* was stored by a call to *waitpid()* that did not specify the
49960 XSI WUNTRACED or WCONTINUED flags, or by a call to the *wait()* function, exactly one of the
49961 macros WIFEXITED(**stat_loc*) and WIFSIGNALLED(**stat_loc*) shall evaluate to a non-zero value.

49962 If the information pointed to by *stat_loc* was stored by a call to *waitpid()* that did not specify the
49963 XSI WUNTRACED flag and specified the WCONTINUED flag, or by a call to the *wait()* function,
49964 XSI exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALLED(**stat_loc*), and
49965 WIFCONTINUED(**stat_loc*) shall evaluate to a non-zero value.

49966 If _POSIX_REALTIME_SIGNALS is defined, and the implementation queues the SIGCHLD
49967 signal, then if *wait()* or *waitpid()* returns because the status of a child process is available, any
49968 pending SIGCHLD signal associated with the process ID of the child process shall be discarded.
49969 Any other pending SIGCHLD signals shall remain pending.

49970 Otherwise, if SIGCHLD is blocked, if *wait()* or *waitpid()* return because the status of a child
49971 process is available, any pending SIGCHLD signal shall be cleared unless the status of another
49972 child process is available.

49973 For all other conditions, it is unspecified whether child *status* will be available when a SIGCHLD
49974 signal is delivered.

49975 There may be additional implementation-defined circumstances under which *wait()* or *waitpid()*
49976 report *status*. This shall not occur unless the calling process or one of its child processes explicitly
49977 makes use of a non-standard extension. In these cases the interpretation of the reported *status* is
49978 implementation-defined.

49979 XSI If a parent process terminates without waiting for all of its child processes to terminate, the
49980 remaining child processes shall be assigned a new parent process ID corresponding to an
49981 implementation-defined system process.

49982 RETURN VALUE

49983 If *wait()* or *waitpid()* returns because the status of a child process is available, these functions
49984 shall return a value equal to the process ID of the child process for which *status* is reported. If
49985 *wait()* or *waitpid()* returns due to the delivery of a signal to the calling process, -1 shall be
49986 returned and *errno* set to [EINTR]. If *waitpid()* was invoked with WNOHANG set in *options*, it
49987 has at least one child process specified by *pid* for which *status* is not available, and *status* is not
49988 available for any process specified by *pid*, 0 is returned. Otherwise, (**pid_t**)-1 shall be returned,
49989 and *errno* set to indicate the error.

49990 ERRORS

49991 The *wait()* function shall fail if:

49992 [ECHILD] The calling process has no existing unwaited-for child processes.

49993 [EINTR] The function was interrupted by a signal. The value of the location pointed to
49994 by *stat_loc* is undefined.

49995 The *waitpid()* function shall fail if:

49996 [ECHILD] The process specified by *pid* does not exist or is not a child of the calling
49997 process, or the process group specified by *pid* does not exist or does not have
49998 any member process that is a child of the calling process.

49999	[EINTR]	The function was interrupted by a signal. The value of the location pointed to by <i>stat_loc</i> is undefined.
50001	[EINVAL]	The <i>options</i> argument is not valid.

50002 EXAMPLES

50003	Waiting for a Child Process and then Checking its Status	2
50004	The following example demonstrates the use of <i>waitpid()</i> , <i>fork()</i> , and the macros used to	2
50005	interpret the status value returned by <i>waitpid()</i> (and <i>wait()</i>). The code segment creates a child	2
50006	process which does some unspecified work. Meanwhile the parent loops performing calls to	2
50007	<i>waitpid()</i> to monitor the status of the child. The loop terminates when child termination is	2
50008	detected.	2
50009	#include <stdio.h>	2
50010	#include <stdlib.h>	2
50011	#include <unistd.h>	2
50012	#include <sys/wait.h>	2
50013	...	2
50014	pid_t child_pid, wpid;	2
50015	int status;	2
50016	child_pid = fork();	2
50017	if (child_pid == -1) { /* fork() failed */	2
50018	perror("fork");	2
50019	exit(EXIT_FAILURE);	2
50020	}	2
50021	if (child_pid == 0) { /* This is the child */	2
50022	/* Child does some work and then terminates */	2
50023	...	2
50024	} else { /* This is the parent */	2
50025	do {	2
50026	wpid = waitpid(child_pid, &status, WUNTRACED	2
50027	#endif WCONTINUED /* Not all implementations support this */	2
50028	WCONTINUED	2
50029	#endif	2
50030);	2
50031	if (wpid == -1) {	2
50032	perror("waitpid");	2
50033	exit(EXIT_FAILURE);	2
50034	}	2
50035	if (WIFEXITED(status)) {	2
50036	printf("child exited, status=%d\n", WEXITSTATUS(status));	2
50037	} else if (WIFSIGNALED(status)) {	2
50038	printf("child killed (signal %d)\n", WTERMSIG(status));	2
50039	} else if (WIFSTOPPED(status)) {	2
50040	printf("child stopped (signal %d)\n", WSTOPSIG(status));	2
50041	#endif WIFCONTINUED /* Not all implementations support this */	2
50042	} else if (WIFCONTINUED(status)) {	2
50043	printf("child continued\n");	2

```

50044     #endif
50045         } else { /* Non-standard case -- may never happen */
50046             printf("Unexpected status (0x%x)\n", status);
50047         }
50048     } while (!WIFEXITED(status) && !WIFSIGNALED(status));
50049 }
```

2
2
2
2
2
2

50050 APPLICATION USAGE

50051 None.

50052 RATIONALE

50053 A call to the *wait()* or *waitpid()* function only returns *status* on an immediate child process of the
 50054 calling process; that is, a child that was produced by a single *fork()* call (perhaps followed by an
 50055 *exec* or other function calls) from the parent. If a child produces grandchildren by further use of
 50056 *fork()*, none of those grandchildren nor any of their descendants affect the behavior of a *wait()*
 50057 from the original parent process. Nothing in this volume of IEEE Std 1003.1-2001 prevents an
 50058 implementation from providing extensions that permit a process to get *status* from a grandchild
 50059 or any other process, but a process that does not use such extensions must be guaranteed to see
 50060 *status* from only its direct children.

50061 The *waitpid()* function is provided for three reasons:

- 50062 1. To support job control
- 50063 2. To permit a non-blocking version of the *wait()* function
- 50064 3. To permit a library routine, such as *system()* or *pclose()*, to wait for its children without
 50065 interfering with other terminated children for which the process has not waited

50066 The first two of these facilities are based on the *wait3()* function provided by 4.3 BSD. The
 50067 function uses the *options* argument, which is equivalent to an argument to *wait3()*. The
 50068 WUNTRACED flag is used only in conjunction with job control on systems supporting job
 50069 control. Its name comes from 4.3 BSD and refers to the fact that there are two types of stopped
 50070 processes in that implementation: processes being traced via the *ptrace()* debugging facility and
 50071 (untraced) processes stopped by job control signals. Since *ptrace()* is not part of this volume of
 50072 IEEE Std 1003.1-2001, only the second type is relevant. The name WUNTRACED was retained
 50073 because its usage is the same, even though the name is not intuitively meaningful in this context.

50074 The third reason for the *waitpid()* function is to permit independent sections of a process to
 50075 spawn and wait for children without interfering with each other. For example, the following
 50076 problem occurs in developing a portable shell, or command interpreter:

```

50077 stream = popen( "/bin/true" );
50078 (void) system( "sleep 100" );
50079 (void) pclose(stream);
```

50080 On all historical implementations, the final *pclose()* fails to reap the *wait()* *status* of the *popen()*.

50081 The status values are retrieved by macros, rather than given as specific bit encodings as they are
 50082 in most historical implementations (and thus expected by existing programs). This was
 50083 necessary to eliminate a limitation on the number of signals an implementation can support that
 50084 was inherent in the traditional encodings. This volume of IEEE Std 1003.1-2001 does require that
 50085 a *status* value of zero corresponds to a process calling *_exit(0)*, as this is the most common
 50086 encoding expected by existing programs. Some of the macro names were adopted from 4.3 BSD.

50087 These macros syntactically operate on an arbitrary integer value. The behavior is undefined
 50088 unless that value is one stored by a successful call to *wait()* or *waitpid()* in the location pointed
 50089 to by the *stat_loc* argument. An early proposal attempted to make this clearer by specifying each

50090 argument as `*stat_loc` rather than `stat_val`. However, that did not follow the conventions of other
50091 specifications in this volume of IEEE Std 1003.1-2001 or traditional usage. It also could have
50092 implied that the argument to the macro must literally be `*stat_loc`; in fact, that value can be
50093 stored or passed as an argument to other functions before being interpreted by these macros.

50094 The extension that affects `wait()` and `waitpid()` and is common in historical implementations is
50095 the `ptrace()` function. It is called by a child process and causes that child to stop and return a
50096 *status* that appears identical to the *status* indicated by `WIFSTOPPED`. The *status* of `ptrace()`
50097 children is traditionally returned regardless of the `WUNTRACED` flag (or by the `wait()`
50098 function). Most applications do not need to concern themselves with such extensions because
50099 they have control over what extensions they or their children use. However, applications, such
50100 as command interpreters, that invoke arbitrary processes may see this behavior when those
50101 arbitrary processes misuse such extensions.

50102 Implementations that support `core` file creation or other implementation-defined actions on
50103 termination of some processes traditionally provide a bit in the *status* returned by `wait()` to
50104 indicate that such actions have occurred.

50105 Allowing the `wait()` family of functions to discard a pending `SIGCHLD` signal that is associated
50106 with a successfully waited-for child process puts them into the `sigwait()` and `sigwaitinfo()`
50107 category with respect to `SIGCHLD`.

50108 This definition allows implementations to treat a pending `SIGCHLD` signal as accepted by the
50109 process in `wait()`, with the same meaning of “accepted” as when that word is applied to the
50110 `sigwait()` family of functions.

50111 Allowing the `wait()` family of functions to behave this way permits an implementation to be able
50112 to deal precisely with `SIGCHLD` signals.

50113 In particular, an implementation that does accept (discard) the `SIGCHLD` signal can make the
50114 following guarantees regardless of the queuing depth of signals in general (the list of waitable
50115 children can hold the `SIGCHLD` queue):

- 50116 1. If a `SIGCHLD` signal handler is established via `sigaction()` without the `SA_RESETHAND`
50117 flag, `SIGCHLD` signals can be accurately counted; that is, exactly one `SIGCHLD` signal will
50118 be delivered to or accepted by the process for every child process that terminates.
- 50119 2. A single `wait()` issued from a `SIGCHLD` signal handler can be guaranteed to return
50120 immediately with status information for a child process.
- 50121 3. When `SA_SIGINFO` is requested, the `SIGCHLD` signal handler can be guaranteed to
50122 receive a non-NULL pointer to a `siginfo_t` structure that describes a child process for
50123 which a wait via `waitpid()` or `waitid()` will not block or fail.
- 50124 4. The `system()` function will not cause a process’ `SIGCHLD` handler to be called as a result of
50125 the `fork()`/`exec` executed within `system()` because `system()` will accept the `SIGCHLD` signal
50126 when it performs a `waitpid()` for its child process. This is a desirable behavior of `system()`
50127 so that it can be used in a library without causing side effects to the application linked with
50128 the library.

50129 An implementation that does not permit the `wait()` family of functions to accept (discard) a
50130 pending `SIGCHLD` signal associated with a successfully waited-for child, cannot make the
50131 guarantees described above for the following reasons:

50132 Guarantee #1

50133 Although it might be assumed that reliable queuing of all `SIGCHLD` signals generated by
50134 the system can make this guarantee, the counter-example is the case of a process that blocks
50135 `SIGCHLD` and performs an indefinite loop of `fork() / wait()` operations. If the

50136 implementation supports queued signals, then eventually the system will run out of
50137 memory for the queue. The guarantee cannot be made because there must be some limit to
50138 the depth of queuing.

50139 **Guarantees #2 and #3**

50140 These cannot be guaranteed unless the *wait()* family of functions accepts the SIGCHLD
50141 signal. Otherwise, a *fork()*/*wait()* executed while SIGCHLD is blocked (as in the *system()*
50142 function) will result in an invocation of the handler when SIGCHLD is unblocked, after the
50143 process has disappeared.

50144 **Guarantee #4**

50145 Although possible to make this guarantee, *system()* would have to set the SIGCHLD
50146 handler to SIG_DFL so that the SIGCHLD signal generated by its *fork()* would be discarded
50147 (the SIGCHLD default action is to be ignored), then restore it to its previous setting. This
50148 would have the undesirable side effect of discarding all SIGCHLD signals pending to the
50149 process.

50150 **FUTURE DIRECTIONS**

50151 None.

50152 **SEE ALSO**

50153 *exec*, *exit()*, *fork()*, *waitid()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**signal.h**>,
50154 <**sys/wait.h**>

50155 **CHANGE HISTORY**

50156 First released in Issue 1. Derived from Issue 1 of the SVID.

50157 **Issue 5**

50158 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

50159 **Issue 6**

50160 The following new requirements on POSIX implementations derive from alignment with the
50161 Single UNIX Specification:

- 50162 • The requirement to include <**sys/types.h**> has been removed. Although <**sys/types.h**> was
50163 required for conforming implementations of previous POSIX specifications, it was not
50164 required for UNIX applications.

50165 The following changes were made to align with the IEEE P1003.1a draft standard:

- 50166 • The processing of the SIGCHLD signal and the [ECHILD] error is clarified.

50167 The semantics of WIFSTOPPED(*stat_val*), WIFEXITED(*stat_val*), and WIFSIGNALED(*stat_val*)
50168 are defined with respect to *posix_spawn()* or *posix_spawnp()* for alignment with
50169 IEEE Std 1003.1d-1999.

50170 The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

50171 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/145 is applied, adding the example to the 2
50172 EXAMPLES section. 2

50173 NAME

50174 waitid — wait for a child process to change state

50175 SYNOPSIS

50176 XSI

```
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

50178

50179 DESCRIPTION

50180 The *waitid()* function shall suspend the calling thread until one child of the process containing
 50181 the calling thread changes state. It records the current state of a child in the structure pointed to
 50182 by *infop*. If a child process changed state prior to the call to *waitid()*, *waitid()* shall return
 50183 immediately. If more than one thread is suspended in *wait()* or *waitpid()* waiting for termination
 50184 of the same process, exactly one thread shall return the process status at the time of the target
 50185 process termination.

50186 The *idtype* and *id* arguments are used to specify which children *waitid()* waits for.

50187 If *idtype* is P_PID, *waitid()* shall wait for the child with a process ID equal to (**pid_t**)*id*.

50188 If *idtype* is P_PGID, *waitid()* shall wait for any child with a process group ID equal to (**pid_t**)*id*.

50189 If *idtype* is P_ALL, *waitid()* shall wait for any children and *id* is ignored.

50190 The *options* argument is used to specify which state changes *waitid()* shall wait for. It is formed
 50191 by OR'ing together one or more of the following flags:

50192 WEXITED Wait for processes that have exited.

50193 WSTOPPED Status shall be returned for any child that has stopped upon receipt of a signal.

50194 WCONTINUED Status shall be returned for any child that was stopped and has been
 50195 continued.

50196 WNOHANG Return immediately if there are no children to wait for.

50197 WNOWAIT Keep the process whose status is returned in *infop* in a waitable state. This
 50198 shall not affect the state of the process; the process may be waited for again
 50199 after this call completes.

50200 The application shall ensure that the *infop* argument points to a **siginfo_t** structure. If *waitid()*
 50201 returns because a child process was found that satisfied the conditions indicated by the
 50202 arguments *idtype* and *options*, then the structure pointed to by *infop* shall be filled in by the
 50203 system with the status of the process. The *si_signo* member shall always be equal to SIGCHLD.

50204 RETURN VALUE

50205 If WNOHANG was specified and there are no children to wait for, 0 shall be returned. If *waitid()*
 50206 returns due to the change of state of one of its children, 0 shall be returned. Otherwise, -1 shall
 50207 be returned and *errno* set to indicate the error.

50208 ERRORS

50209 The *waitid()* function shall fail if:

50210 [ECHILD] The calling process has no existing unwaited-for child processes.

50211 [EINTR] The *waitid()* function was interrupted by a signal.

50212 [EINVAL] An invalid value was specified for *options*, or *idtype* and *id* specify an invalid
 50213 set of processes.

50214 EXAMPLES

50215 None.

50216 APPLICATION USAGE

50217 None.

50218 RATIONALE

50219 None.

50220 FUTURE DIRECTIONS

50221 None.

50222 SEE ALSO

50223 *exec*, *exit()*, *wait()*, the Base Definitions volume of IEEE Std 1003.1-2001, <sys/wait.h>

50224 CHANGE HISTORY

50225 First released in Issue 4, Version 2.

50226 Issue 5

50227 Moved from X/OPEN UNIX extension to BASE.

50228 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

50229 Issue 6

50230 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

50231 **NAME**

50232 waitpid — wait for a child process to stop or terminate

50233 **SYNOPSIS**

50234 #include <sys/wait.h>

50235 pid_t waitpid(pid_t *pid*, int **stat_loc*, int *options*);

50236 **DESCRIPTION**

50237 Refer to *wait()*.

50238 NAME

50239 wcrtomb — convert a wide-character code to a character (restartable)

50240 SYNOPSIS

```
50241       #include <stdio.h>
50242       size_t wcrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);
```

50243 DESCRIPTION

50244 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50245 conflict between the requirements described here and the ISO C standard is unintentional. This
50246 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50247 If *s* is a null pointer, the *wcrtomb()* function shall be equivalent to the call:

```
50248       wcrtomb(buf, L'\0', ps)
```

50249 where *buf* is an internal buffer.

50250 If *s* is not a null pointer, the *wcrtomb()* function shall determine the number of bytes needed to
50251 represent the character that corresponds to the wide character given by *wc* (including any shift
50252 sequences), and store the resulting bytes in the array whose first element is pointed to by *s*. At
50253 most {MB_CUR_MAX} bytes are stored. If *wc* is a null wide character, a null byte shall be stored,
50254 preceded by any shift sequence needed to restore the initial shift state. The resulting state
50255 described shall be the initial conversion state.

50256 If *ps* is a null pointer, the *wcrtomb()* function shall use its own internal **mbstate_t** object, which is
50257 initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object
50258 pointed to by *ps* shall be used to completely describe the current conversion state of the
50259 associated character sequence. The implementation shall behave as if no function defined in this
50260 volume of IEEE Std 1003.1-2001 calls *wcrtomb()*.

50261 CX If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS
50262 functions, the application shall ensure that the *wcrtomb()* function is called with a non-NUL *ps*
50263 argument.

50264 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

50265 RETURN VALUE

50266 The *wcrtomb()* function shall return the number of bytes stored in the array object (including any
50267 shift sequences). When *wc* is not a valid wide character, an encoding error shall occur. In this
50268 case, the function shall store the value of the macro [EILSEQ] in *errno* and shall return (**size_t**)-1;
50269 the conversion state shall be undefined.

50270 ERRORS

50271 The *wcrtomb()* function may fail if:

50272 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

50273 [EILSEQ] Invalid wide-character code is detected.

50274 EXAMPLES

50275 None.

50276 APPLICATION USAGE

50277 None.

50278 RATIONALE

50279 None.

50280 FUTURE DIRECTIONS

50281 None.

50282 SEE ALSO

50283 *mbsinit()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

50284 CHANGE HISTORY

50285 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50286 (E).

50287 Issue 6

50288 In the DESCRIPTION, a note on using this function in a threaded application is added.
50289 Extensions beyond the ISO C standard are marked.
50290 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
50291 The *wcrtomb()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

50292 NAME

50293 **wcscat** — concatenate two wide-character strings

50294 SYNOPSIS

```
50295        #include <wchar.h>
50296        wchar_t *wcscat(wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

50297 DESCRIPTION

50298 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50299 conflict between the requirements described here and the ISO C standard is unintentional. This
50300 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50301 The **wcscat()** function shall append a copy of the wide-character string pointed to by *ws2*
50302 (including the terminating null wide-character code) to the end of the wide-character string
50303 pointed to by *ws1*. The initial wide-character code of *ws2* shall overwrite the null wide-character
50304 code at the end of *ws1*. If copying takes place between objects that overlap, the behavior is
50305 undefined.

50306 RETURN VALUE

50307 The **wcscat()** function shall return *ws1*; no return value is reserved to indicate an error.

50308 ERRORS

50309 No errors are defined.

50310 EXAMPLES

50311 None.

50312 APPLICATION USAGE

50313 None.

50314 RATIONALE

50315 None.

50316 FUTURE DIRECTIONS

50317 None.

50318 SEE ALSO

50319 **wcsncat()**, the Base Definitions volume of IEEE Std 1003.1-2001, <**wchar.h**>

50320 CHANGE HISTORY

50321 First released in Issue 4. Derived from the MSE working draft.

50322 Issue 6

50323 The Open Group Corrigendum U040/2 is applied. In the RETURN VALUE section, *s1* is changed
50324 to *ws1*.

50325 The **wcscat()** prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

50326 NAME

50327 wcschr — wide-character string scanning operation

50328 SYNOPSIS

50329 #include <wchar.h>
50330 wchar_t *wcschr(const wchar_t *ws, wchar_t wc);

50331 DESCRIPTION

50332 CX 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.

50335 The *wcschr()* function shall locate the first occurrence of *wc* in the wide-character string pointed to by *ws*. The application shall ensure that the value of *wc* is a character representable as a type **wchar_t** and a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide-character string.

50339 RETURN VALUE

50340 Upon completion, *wcschr()* shall return a pointer to the wide-character code, or a null pointer if
50341 the wide-character code is not found.

50342 ERRORS

50343 No errors are defined.

50344 EXAMPLES

50345 None.

50346 APPLICATION USAGE

50347 None.

50348 RATIONALE

50349 None.

50350 FUTURE DIRECTIONS

50351 None.

50352 SEE ALSO

50353 *wcsrchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

50354 CHANGE HISTORY

50355 First released in Issue 4. Derived from the MSE working draft.

50356 Issue 6

50357 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

50358 NAME

50359 `wcscmp` — compare two wide-character strings

50360 SYNOPSIS

```
50361        #include <wchar.h>
50362        int wcscmp(const wchar_t *ws1, const wchar_t *ws2);
```

50363 DESCRIPTION

50364 CX 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.

50367 The `wcscmp()` function shall compare the wide-character string pointed to by `ws1` to the wide-character string pointed to by `ws2`.

50369 The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared.

50371 RETURN VALUE

50372 Upon completion, `wcscmp()` shall return an integer greater than, equal to, or less than 0, if the wide-character string pointed to by `ws1` is greater than, equal to, or less than the wide-character string pointed to by `ws2`, respectively.

50375 ERRORS

50376 No errors are defined.

50377 EXAMPLES

50378 None.

50379 APPLICATION USAGE

50380 None.

50381 RATIONALE

50382 None.

50383 FUTURE DIRECTIONS

50384 None.

50385 SEE ALSO

50386 `wcsncmp()`, the Base Definitions volume of IEEE Std 1003.1-2001, <`wchar.h`>

50387 CHANGE HISTORY

50388 First released in Issue 4. Derived from the MSE working draft.

50389 NAME

50390 wcscoll — wide-character string comparison using collating information

50391 SYNOPSIS

```
50392       #include <wchar.h>
50393       int wcscoll(const wchar_t *ws1, const wchar_t *ws2);
```

50394 DESCRIPTION

50395 CX 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.

50398 The *wcscoll()* function shall compare the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*, both interpreted as appropriate to the *LC_COLLATE* category of the current locale.

50401 CX The *wcscoll()* function shall not change the setting of *errno* if successful.

50402 An application wishing to check for error situations should set *errno* to 0 before calling *wcscoll()*. If *errno* is non-zero on return, an error has occurred.

50404 RETURN VALUE

50405 Upon successful completion, *wcscoll()* shall return an integer greater than, equal to, or less than 0, according to whether the wide-character string pointed to by *ws1* is greater than, equal to, or less than the wide-character string pointed to by *ws2*, when both are interpreted as appropriate to the current locale. On error, *wcscoll()* shall set *errno*, but no return value is reserved to indicate an error.

50410 ERRORS

50411 The *wcscoll()* function may fail if:

50412 CX [EINVAL] The *ws1* or *ws2* arguments contain wide-character codes outside the domain of the collating sequence.

50414 EXAMPLES

50415 None.

50416 APPLICATION USAGE

50417 The *wcsxfrm()* and *wcscmp()* functions should be used for sorting large lists.

50418 RATIONALE

50419 None.

50420 FUTURE DIRECTIONS

50421 None.

50422 SEE ALSO

50423 *wcscmp()*, *wcsxfrm()*, the Base Definitions volume of IEEE Std 1003.1-2001, <*wchar.h*>

50424 CHANGE HISTORY

50425 First released in Issue 4. Derived from the MSE working draft.

50426 Issue 5

50427 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

50428 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

50429 NAME

50430 `wcscpy` — copy a wide-character string

50431 SYNOPSIS

50432 `#include <wchar.h>`

50433 `wchar_t *wcscpy(wchar_t *restrict ws1, const wchar_t *restrict ws2);`

50434 DESCRIPTION

50435 CX 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.

50438 The `wcscpy()` function shall copy the wide-character string pointed to by `ws2` (including the terminating null wide-character code) into the array pointed to by `ws1`. If copying takes place between objects that overlap, the behavior is undefined.

50441 RETURN VALUE

50442 The `wcscpy()` function shall return `ws1`; no return value is reserved to indicate an error.

50443 ERRORS

50444 No errors are defined.

50445 EXAMPLES

50446 None.

50447 APPLICATION USAGE

50448 None.

50449 RATIONALE

50450 None.

50451 FUTURE DIRECTIONS

50452 None.

50453 SEE ALSO

50454 `wcsncpy()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`

50455 CHANGE HISTORY

50456 First released in Issue 4. Derived from the MSE working draft.

50457 Issue 6

50458 The `wcscpy()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

50459 NAME

50460 `wcscspn` — get the length of a complementary wide substring

50461 SYNOPSIS

```
50462        #include <wchar.h>
50463        size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);
```

50464 DESCRIPTION

50465 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50466 conflict between the requirements described here and the ISO C standard is unintentional. This
50467 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50468 The `wcscspn()` function shall compute the length (in wide characters) of the maximum initial
50469 segment of the wide-character string pointed to by `ws1` which consists entirely of wide-character
50470 codes *not* from the wide-character string pointed to by `ws2`.

50471 RETURN VALUE

50472 The `wcscspn()` function shall return the length of the initial substring of `ws1`; no return value is
50473 reserved to indicate an error.

50474 ERRORS

50475 No errors are defined.

50476 EXAMPLES

50477 None.

50478 APPLICATION USAGE

50479 None.

50480 RATIONALE

50481 None.

50482 FUTURE DIRECTIONS

50483 None.

50484 SEE ALSO

50485 `wcsspn()`, the Base Definitions volume of IEEE Std 1003.1-2001, <`wchar.h`>

50486 CHANGE HISTORY

50487 First released in Issue 4. Derived from the MSE working draft.

50488 Issue 5

50489 The RETURN VALUE section is updated to indicate that `wcscspn()` returns the length of `ws1`,
50490 rather than `ws1` itself.

50491 NAME

50492 wcsftime — convert date and time to a wide-character string

50493 SYNOPSIS

```
50494        #include <wchar.h>
50495        size_t wcsftime(wchar_t *restrict wcs, size_t maxsize,
50496                    const wchar_t *restrict format, const struct tm *restrict timeptr);
```

50497 DESCRIPTION

50498 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50499 conflict between the requirements described here and the ISO C standard is unintentional. This
50500 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50501 The *wcsftime()* function shall be equivalent to the *strftime()* function, except that:

- 50502 • The argument *wcs* points to the initial element of an array of wide characters into which the
50503 generated output is to be placed.
- 50504 • The argument *maxsize* indicates the maximum number of wide characters to be placed in the
50505 output array.
- 50506 • The argument *format* is a wide-character string and the conversion specifications are replaced
50507 by corresponding sequences of wide characters.
- 50508 • The return value indicates the number of wide characters placed in the output array.

50509 If copying takes place between objects that overlap, the behavior is undefined.

50510 RETURN VALUE

50511 If the total number of resulting wide-character codes including the terminating null wide-
50512 character code is no more than *maxsize*, *wcsftime()* shall return the number of wide-character
50513 codes placed into the array pointed to by *wcs*, not including the terminating null wide-character
50514 code. Otherwise, zero is returned and the contents of the array are unspecified.

50515 ERRORS

50516 No errors are defined.

50517 EXAMPLES

50518 None.

50519 APPLICATION USAGE

50520 None.

50521 RATIONALE

50522 None.

50523 FUTURE DIRECTIONS

50524 None.

50525 SEE ALSO

50526 *strftime()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

50527 CHANGE HISTORY

50528 First released in Issue 4.

50529 Issue 5

50530 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

50531 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of the *format*
50532 argument is changed from **const char *** to **const wchar_t ***.

50533 Issue 6

50534

The *wcsftime()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

50535 NAME

50536 `wcslen — get wide-character string length`

50537 SYNOPSIS

50538 `#include <wchar.h>`
50539 `size_t wcslen(const wchar_t *ws);`

50540 DESCRIPTION

50541 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50542 conflict between the requirements described here and the ISO C standard is unintentional. This
50543 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50544 The `wcslen()` function shall compute the number of wide-character codes in the wide-character
50545 string to which `ws` points, not including the terminating null wide-character code.

50546 RETURN VALUE

50547 The `wcslen()` function shall return the length of `ws`; no return value is reserved to indicate an
50548 error.

50549 ERRORS

50550 No errors are defined.

50551 EXAMPLES

50552 None.

50553 APPLICATION USAGE

50554 None.

50555 RATIONALE

50556 None.

50557 FUTURE DIRECTIONS

50558 None.

50559 SEE ALSO

50560 The Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`

50561 CHANGE HISTORY

50562 First released in Issue 4. Derived from the MSE working draft.

50563 NAME

50564 `wcsncat` — concatenate a wide-character string with part of another

50565 SYNOPSIS

```
50566        #include <wchar.h>
50567        wchar_t *wcsncat(wchar_t *restrict ws1, const wchar_t *restrict ws2,
50568                    size_t n);
```

50569 DESCRIPTION

50570 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50571 conflict between the requirements described here and the ISO C standard is unintentional. This
50572 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50573 The `wcsncat()` function shall append not more than *n* wide-character codes (a null wide-
50574 character code and wide-character codes that follow it are not appended) from the array pointed
50575 to by *ws2* to the end of the wide-character string pointed to by *ws1*. The initial wide-character
50576 code of *ws2* shall overwrite the null wide-character code at the end of *ws1*. A terminating null
50577 wide-character code shall always be appended to the result. If copying takes place between
50578 objects that overlap, the behavior is undefined.

50579 RETURN VALUE

50580 The `wcsncat()` function shall return *ws1*; no return value is reserved to indicate an error.

50581 ERRORS

50582 No errors are defined.

50583 EXAMPLES

50584 None.

50585 APPLICATION USAGE

50586 None.

50587 RATIONALE

50588 None.

50589 FUTURE DIRECTIONS

50590 None.

50591 SEE ALSO

50592 `wcscat()`, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

50593 CHANGE HISTORY

50594 First released in Issue 4. Derived from the MSE working draft.

50595 Issue 6

50596 The `wcsncat()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

50597 NAME

50598 `wcsncmp` — compare part of two wide-character strings

50599 SYNOPSIS

```
50600        #include <wchar.h>
50601        int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

50602 DESCRIPTION

50603 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50604 conflict between the requirements described here and the ISO C standard is unintentional. This
50605 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50606 The `wcsncmp()` function shall compare not more than *n* wide-character codes (wide-character
50607 codes that follow a null wide-character code are not compared) from the array pointed to by *ws1*
50608 to the array pointed to by *ws2*.

50609 The sign of a non-zero return value shall be determined by the sign of the difference between the
50610 values of the first pair of wide-character codes that differ in the objects being compared.

50611 RETURN VALUE

50612 Upon successful completion, `wcsncmp()` shall return an integer greater than, equal to, or less
50613 than 0, if the possibly null-terminated array pointed to by *ws1* is greater than, equal to, or less
50614 than the possibly null-terminated array pointed to by *ws2*, respectively.

50615 ERRORS

50616 No errors are defined.

50617 EXAMPLES

50618 None.

50619 APPLICATION USAGE

50620 None.

50621 RATIONALE

50622 None.

50623 FUTURE DIRECTIONS

50624 None.

50625 SEE ALSO

50626 `wcscmp()`, the Base Definitions volume of IEEE Std 1003.1-2001, <`wchar.h`>

50627 CHANGE HISTORY

50628 First released in Issue 4. Derived from the MSE working draft.

50629 NAME

50630 wcsncpy — copy part of a wide-character string

50631 SYNOPSIS

```
50632        #include <wchar.h>
50633        wchar_t *wcsncpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,
50634                    size_t n);
```

50635 DESCRIPTION

50636 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50637 conflict between the requirements described here and the ISO C standard is unintentional. This
50638 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50639 The `wcsncpy()` function shall copy not more than *n* wide-character codes (wide-character codes
50640 that follow a null wide-character code are not copied) from the array pointed to by *ws2* to the
50641 array pointed to by *ws1*. If copying takes place between objects that overlap, the behavior is
50642 undefined.

50643 If the array pointed to by *ws2* is a wide-character string that is shorter than *n* wide-character
50644 codes, null wide-character codes shall be appended to the copy in the array pointed to by *ws1*,
50645 until *n* wide-character codes in all are written.

50646 RETURN VALUE

50647 The `wcsncpy()` function shall return *ws1*; no return value is reserved to indicate an error.

50648 ERRORS

50649 No errors are defined.

50650 EXAMPLES

50651 None.

50652 APPLICATION USAGE

50653 If there is no null wide-character code in the first *n* wide-character codes of the array pointed to
50654 by *ws2*, the result is not null-terminated.

50655 RATIONALE

50656 None.

50657 FUTURE DIRECTIONS

50658 None.

50659 SEE ALSO

50660 `wcscpy()`, the Base Definitions volume of IEEE Std 1003.1-2001, <`wchar.h`>

50661 CHANGE HISTORY

50662 First released in Issue 4. Derived from the MSE working draft.

50663 Issue 6

50664 The `wcsncpy()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

50665 NAME

50666 *wcspbrk* — scan a wide-character string for a wide-character code

50667 SYNOPSIS

```
50668        #include <wchar.h>
50669        wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
```

50670 DESCRIPTION

50671 CX 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.

50674 The *wcspbrk()* function shall locate the first occurrence in the wide-character string pointed to by *ws1* of any wide-character code from the wide-character string pointed to by *ws2*.

50676 RETURN VALUE

50677 Upon successful completion, *wcspbrk()* shall return a pointer to the wide-character code or a null pointer if no wide-character code from *ws2* occurs in *ws1*.

50679 ERRORS

50680 No errors are defined.

50681 EXAMPLES

50682 None.

50683 APPLICATION USAGE

50684 None.

50685 RATIONALE

50686 None.

50687 FUTURE DIRECTIONS

50688 None.

50689 SEE ALSO

50690 *wcschr()*, *wcsrchr()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**wchar.h**>

50691 CHANGE HISTORY

50692 First released in Issue 4. Derived from the MSE working draft.

50693 NAME

50694 `wcsrchr` — wide-character string scanning operation

50695 SYNOPSIS

```
50696        #include <wchar.h>
50697        wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);
```

50698 DESCRIPTION

50699 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50700 conflict between the requirements described here and the ISO C standard is unintentional. This
50701 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50702 The `wcsrchr()` function shall locate the last occurrence of `wc` in the wide-character string pointed
50703 to by `ws`. The application shall ensure that the value of `wc` is a character representable as a type
50704 `wchar_t` and a wide-character code corresponding to a valid character in the current locale. The
50705 terminating null wide-character code shall be considered to be part of the wide-character string.

50706 RETURN VALUE

50707 Upon successful completion, `wcsrchr()` shall return a pointer to the wide-character code or a null
50708 pointer if `wc` does not occur in the wide-character string.

50709 ERRORS

50710 No errors are defined.

50711 EXAMPLES

50712 None.

50713 APPLICATION USAGE

50714 None.

50715 RATIONALE

50716 None.

50717 FUTURE DIRECTIONS

50718 None.

50719 SEE ALSO

50720 `weschr()`, the Base Definitions volume of IEEE Std 1003.1-2001, <`wchar.h`>

50721 CHANGE HISTORY

50722 First released in Issue 4. Derived from the MSE working draft.

50723 Issue 6

50724 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

50725 NAME

50726 wcsrtombs — convert a wide-character string to a character string (restartable)

50727 SYNOPSIS

```
50728     #include <wchar.h>
50729
50730     size_t wcsrtombs(char *restrict dst, const wchar_t **restrict src,
50731                     size_t len, mbstate_t *restrict ps);
```

50731 DESCRIPTION

50732 CX 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.

50735 The *wcsrtombs()* function shall convert a sequence of wide characters from the array indirectly pointed to by *src* into a sequence of corresponding characters, beginning in the conversion state described by the object pointed to by *ps*. If *dst* is not a null pointer, the converted characters shall then be stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null wide character, which shall also be stored. Conversion shall stop earlier in the following cases:

- When a code is reached that does not correspond to a valid character
- When the next character would exceed the limit of *len* total bytes to be stored in the array pointed to by *dst* (and *dst* is not a null pointer)

50744 Each conversion shall take place as if by a call to the *wcrtomb()* function.

50745 If *dst* is not a null pointer, the pointer object pointed to by *src* shall be assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described shall be the initial conversion state.

50749 If *ps* is a null pointer, the *wcsrtombs()* function shall use its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* shall be used to completely describe the current conversion state of the associated character sequence. The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-2001 calls *wcsrtombs()*.

50754 CX If the application uses any of the **_POSIX_THREAD_SAFE_FUNCTIONS** or **_POSIX_THREADS** functions, the application shall ensure that the *wcsrtombs()* function is called with a non-NUL *ps* argument.

50757 The behavior of this function shall be affected by the **LC_CTYPE** category of the current locale.

50758 RETURN VALUE

50759 If conversion stops because a code is reached that does not correspond to a valid character, an encoding error occurs. In this case, the *wcsrtombs()* function shall store the value of the macro **[EILSEQ]** in *errno* and return (*size_t*)–1; the conversion state is undefined. Otherwise, it shall return the number of bytes in the resulting character sequence, not including the terminating null (if any).

50764 ERRORS

50765 The *wcsrtombs()* function may fail if:

50766 CX [EINVAL] *ps* points to an object that contains an invalid conversion state.

50767 [EILSEQ] A wide-character code does not correspond to a valid character.

50768 EXAMPLES

50769 None.

50770 APPLICATION USAGE

50771 None.

50772 RATIONALE

50773 None.

50774 FUTURE DIRECTIONS

50775 None.

50776 SEE ALSO

50777 *mbsinit()*, *wcrtomb()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

50778 CHANGE HISTORY

50779 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50780 (E).

50781 Issue 6

50782 In the DESCRIPTION, a note on using this function in a threaded application is added.
50783 Extensions beyond the ISO C standard are marked.
50784 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
50785 The *wcsrtombs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

50786 NAME

50787 `wcspn` — get the length of a wide substring

50788 SYNOPSIS

```
50789        #include <wchar.h>
50790        size_t wcspn(const wchar_t *ws1, const wchar_t *ws2);
```

50791 DESCRIPTION

50792 CX 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.

50795 The `wcspn()` function shall compute the length (in wide characters) of the maximum initial segment of the wide-character string pointed to by `ws1` which consists entirely of wide-character codes from the wide-character string pointed to by `ws2`.

50798 RETURN VALUE

50799 The `wcspn()` function shall return the length of the initial substring of `ws1`; no return value is reserved to indicate an error.

50801 ERRORS

50802 No errors are defined.

50803 EXAMPLES

50804 None.

50805 APPLICATION USAGE

50806 None.

50807 RATIONALE

50808 None.

50809 FUTURE DIRECTIONS

50810 None.

50811 SEE ALSO

50812 `wcscspn()`, the Base Definitions volume of IEEE Std 1003.1-2001, <`wchar.h`>

50813 CHANGE HISTORY

50814 First released in Issue 4. Derived from the MSE working draft.

50815 Issue 5

50816 The RETURN VALUE section is updated to indicate that `wcspn()` returns the length of `ws1`
50817 rather than `ws1` itself.

50818 NAME

50819 `wcsstr` — find a wide-character substring

50820 SYNOPSIS

```
50821        #include <wchar.h>
50822        wchar_t *wcsstr(const wchar_t *restrict ws1,
50823                    const wchar_t *restrict ws2);
```

50824 DESCRIPTION

50825 CX The functionality described on this reference page is aligned with the ISO C standard. Any
50826 conflict between the requirements described here and the ISO C standard is unintentional. This
50827 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50828 The `wcsstr()` function shall locate the first occurrence in the wide-character string pointed to by
50829 `ws1` of the sequence of wide characters (excluding the terminating null wide character) in the
50830 wide-character string pointed to by `ws2`.

50831 RETURN VALUE

50832 Upon successful completion, `wcsstr()` shall return a pointer to the located wide-character string,
50833 or a null pointer if the wide-character string is not found.

50834 If `ws2` points to a wide-character string with zero length, the function shall return `ws1`.

50835 ERRORS

50836 No errors are defined.

50837 EXAMPLES

50838 None.

50839 APPLICATION USAGE

50840 None.

50841 RATIONALE

50842 None.

50843 FUTURE DIRECTIONS

50844 None.

50845 SEE ALSO

50846 `wcschr()`, the Base Definitions volume of IEEE Std 1003.1-2001, <`wchar.h`>

50847 CHANGE HISTORY

50848 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
50849 (E).

50850 Issue 6

50851 The `wcsstr()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

50852 NAME

50853 wcstod, wcstof, wcstold — convert a wide-character string to a double-precision number

50854 SYNOPSIS

```
50855       #include <wchar.h>
50856
50857       double wcstod(const wchar_t *restrict nptr, wchar_t **restrict endptr);
50858       float wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);
50859       long double wcstold(const wchar_t *restrict nptr,
50860                        wchar_t **restrict endptr);
```

50860 DESCRIPTION

50861 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 50862 conflict between the requirements described here and the ISO C standard is unintentional. This
 50863 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

50864 These functions shall convert the initial portion of the wide-character string pointed to by *nptr* to
 50865 **double**, **float**, and **long double** representation, respectively. First, they shall decompose the
 50866 input wide-character string into three parts:

- 50867 1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by
 50868 *iswspace*())
- 50869 2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
- 50870 3. A final wide-character string of one or more unrecognized wide-character codes, including
 50871 the terminating null wide-character code of the input wide-character string

50872 Then they shall attempt to convert the subject sequence to a floating-point number, and return
 50873 the result.

50874 The expected form of the subject sequence is an optional plus or minus sign, then one of the
 50875 following:

- 50876 • A non-empty sequence of decimal digits optionally containing a radix character, then an
 50877 optional exponent part
- 50878 • A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix
 50879 character, then an optional binary exponent part
- 50880 • One of INF or INFINITY, or any other wide string equivalent except for case
- 50881 • One of NAN or NAN(*n-wchar-sequence*_{opt}), or any other wide string ignoring case in the NAN
 50882 part, where:

```
50883       n-wchar-sequence:
50884            digit
50885            nondigit
50886            n-wchar-sequence digit
50887            n-wchar-sequence nondigit
```

50888 The subject sequence is defined as the longest initial subsequence of the input wide string,
 50889 starting with the first non-white-space wide character, that is of the expected form. The subject
 50890 sequence contains no wide characters if the input wide string is not of the expected form.

50891 If the subject sequence has the expected form for a floating-point number, the sequence of wide
 50892 characters starting with the first digit or the radix character (whichever occurs first) shall be
 50893 interpreted as a floating constant according to the rules of the C language, except that the radix
 50894 character shall be used in place of a period, and that if neither an exponent part nor a radix
 50895 character appears in a decimal floating-point number, or if a binary exponent part does not

50896 appear in a hexadecimal floating-point number, an exponent part of the appropriate type with
50897 value zero shall be assumed to follow the last digit in the string. If the subject sequence begins
50898 with a minus sign, the sequence shall be interpreted as negated. A wide-character sequence INF
50899 or INFINITY shall be interpreted as an infinity, if representable in the return type, else as if it
50900 were a floating constant that is too large for the range of the return type. A wide-character
50901 sequence NAN or NAN(*n-wchar-sequence_{opt}*) shall be interpreted as a quiet NaN, if supported in
50902 the return type, else as if it were a subject sequence part that does not have the expected form;
50903 the meaning of the *n-wchar* sequences is implementation-defined. A pointer to the final wide
50904 string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

50905 If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the
50906 conversion shall be rounded in an implementation-defined manner.

50907 CX The radix character shall be as defined in the program's locale (category LC_NUMERIC). In the
50908 POSIX locale, or in a locale where the radix character is not defined, the radix character shall
50909 default to a period ('.').

50910 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be
50911 accepted.

50912 If the subject sequence is empty or does not have the expected form, no conversion shall be
50913 performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that
50914 *endptr* is not a null pointer.

50915 CX The *wcstod()* function shall not change the setting of *errno* if successful.

50916 Since 0 is returned on error and is also a valid return on success, an application wishing to check
50917 for error situations should set *errno* to 0, then call *wcstod()*, *wcstof()*, or *wcstold()*, then check
50918 *errno*.

50919 RETURN VALUE

50920 Upon successful completion, these functions shall return the converted value. If no conversion
50921 CX could be performed, 0 shall be returned and *errno* may be set to [EINVAL].

50922 If the correct value is outside the range of representable values, ±HUGE_VAL, ±HUGE_VALF, or 1
50923 ±HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to 1
50924 [ERANGE].

50925 If the correct value would cause underflow, a value whose magnitude is no greater than the
50926 smallest normalized positive number in the return type shall be returned and *errno* set to
50927 [ERANGE].

50928 ERRORS

50929 The *wcstod()* function shall fail if:

50930 [ERANGE] The value to be returned would cause overflow or underflow.

50931 The *wcstod()* function may fail if:

50932 CX [EINVAL] No conversion could be performed.

50933 **EXAMPLES**

50934 None.

50935 **APPLICATION USAGE**

50936 If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, and the
50937 result is not exactly representable, the result should be one of the two numbers in the
50938 appropriate internal format that are adjacent to the hexadecimal floating source value, with the
50939 extra stipulation that the error should have a correct sign for the current rounding direction.

50940 If the subject sequence has the decimal form and at most DECIMAL_DIG (defined in <float.h>)
50941 significant digits, the result should be correctly rounded. If the subject sequence *D* has the
50942 decimal form and more than DECIMAL_DIG significant digits, consider the two bounding,
50943 adjacent decimal strings *L* and *U*, both having DECIMAL_DIG significant digits, such that the
50944 values of *L*, *D*, and *U* satisfy "*L* <= *D* <= *U*". The result should be one of the (equal or
50945 adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current
50946 rounding direction, with the extra stipulation that the error with respect to *D* should have a
50947 correct sign for the current rounding direction.

50948 **RATIONALE**

50949 None.

50950 **FUTURE DIRECTIONS**

50951 None.

50952 **SEE ALSO**

50953 *iswspace()*, *localeconv()*, *scanf()*, *setlocale()*, *wcstol()*, the Base Definitions volume of
50954 IEEE Std 1003.1-2001, Chapter 7, Locale, <float.h>, <wchar.h>

50955 **CHANGE HISTORY**

50956 First released in Issue 4. Derived from the MSE working draft.

50957 **Issue 5**

50958 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

50959 **Issue 6**

50960 Extensions beyond the ISO C standard are marked.

50961 The following new requirements on POSIX implementations derive from alignment with the
50962 Single UNIX Specification:

- 50963 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
50964 added if no conversion could be performed.

50965 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 50966 • The *wcstod()* prototype is updated.
- 50967 • The *wcstof()* and *wcstold()* functions are added.
- 50968 • If the correct value for *wcstod()* would cause underflow, the return value changed from 0 (as
50969 specified in Issue 5) to the smallest normalized positive number.
- 50970 • The DESCRIPTION, RETURN VALUE, and APPLICATION USAGE sections are extensively
50971 updated.

50972 ISO/IEC 9899:1999 standard, Technical Corrigendum 1 is incorporated.

50973 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/66 is applied, correcting the second 1
50974 paragraph in the RETURN VALUE section. 1

50975 NAME

50976 wcstoimax, wcstoumax — convert a wide-character string to an integer type

50977 SYNOPSIS

```
50978       #include <stddef.h>
50979       #include <inttypes.h>
50980       intmax_t wcstoimax(const wchar_t *restrict nptr,
50981                    wchar_t **restrict endptr, int base);
50982       uintmax_t wcstoumax(const wchar_t *restrict nptr,
50983                    wchar_t **restrict endptr, int base);
```

50984 DESCRIPTION

50985 CX 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.

50988 These functions shall be equivalent to the *wcstol()*, *wcstoll()*, *wcstoul()*, and *wcstoull()* functions, respectively, except that the initial portion of the wide string shall be converted to **intmax_t** and **uintmax_t** representation, respectively.

50991 RETURN VALUE

50992 These functions shall return the converted value, if any.

50993 If no conversion could be performed, zero shall be returned. If the correct value is outside the range of representable values, {INTMAX_MAX}, {INTMAX_MIN}, or {UINTMAX_MAX} shall be returned (according to the return type and sign of the value, if any), and *errno* shall be set to [ERANGE].

50997 ERRORS

50998 These functions shall fail if:

50999 [EINVAL] The value of *base* is not supported.

51000 [ERANGE] The value to be returned is not representable.

51001 These functions may fail if:

51002 [EINVAL] No conversion could be performed.

51003 EXAMPLES

51004 None.

51005 APPLICATION USAGE

51006 None.

51007 RATIONALE

51008 None.

51009 FUTURE DIRECTIONS

51010 None.

51011 SEE ALSO

51012 *wcstol()*, *wcstoul()*, the Base Definitions volume of IEEE Std 1003.1-2001, <inttypes.h>,
51013 <stddef.h>

51014 CHANGE HISTORY

51015 First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

51016 NAME

51017 wcstok — split a wide-character string into tokens

51018 SYNOPSIS

51019 #include <wchar.h>
51020
51021 wchar_t *wcstok(wchar_t *restrict ws1, const wchar_t *restrict ws2,
51021 wchar_t **restrict ptr);

51022 DESCRIPTION

51023 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51024 conflict between the requirements described here and the ISO C standard is unintentional. This
51025 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

51026 A sequence of calls to *wcstok()* shall break the wide-character string pointed to by *ws1* into a
51027 sequence of tokens, each of which shall be delimited by a wide-character code from the wide-
51028 character string pointed to by *ws2*. The *ptr* argument points to a caller-provided **wchar_t** pointer
51029 into which the *wcstok()* function shall store information necessary for it to continue scanning the
51030 same wide-character string.

51031 The first call in the sequence has *ws1* as its first argument, and is followed by calls with a null
51032 pointer as their first argument. The separator string pointed to by *ws2* may be different from call
51033 to call.

51034 The first call in the sequence shall search the wide-character string pointed to by *ws1* for the first
51035 wide-character code that is *not* contained in the current separator string pointed to by *ws2*. If no
51036 such wide-character code is found, then there are no tokens in the wide-character string pointed
51037 to by *ws1* and *wcstok()* shall return a null pointer. If such a wide-character code is found, it shall
51038 be the start of the first token.

51039 The *wcstok()* function shall then search from there for a wide-character code that *is* contained in
51040 the current separator string. If no such wide-character code is found, the current token extends
51041 to the end of the wide-character string pointed to by *ws1*, and subsequent searches for a token
51042 shall return a null pointer. If such a wide-character code is found, it shall be overwritten by a
51043 null wide character, which terminates the current token. The *wcstok()* function shall save a
51044 pointer to the following wide-character code, from which the next search for a token shall start.

51045 Each subsequent call, with a null pointer as the value of the first argument, shall start searching
51046 from the saved pointer and behave as described above.

51047 The implementation shall behave as if no function calls *wcstok()*.

51048 RETURN VALUE

51049 Upon successful completion, the *wcstok()* function shall return a pointer to the first wide-
51050 character code of a token. Otherwise, if there is no token, *wcstok()* shall return a null pointer.

51051 ERRORS

51052 No errors are defined.

51053 EXAMPLES

51054 None.

51055 APPLICATION USAGE

51056 None.

51057 RATIONALE

51058 None.

51059 FUTURE DIRECTIONS

51060 None.

51061 SEE ALSO

51062 The Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

51063 CHANGE HISTORY

51064 First released in Issue 4.

51065 Issue 5

51066 Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, a third argument is added to the definition of `wcstok()` in the SYNOPSIS.

51068 Issue 6

51069 The `wcstok()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

51070 NAME

51071 wcstol, wcstoll — convert a wide-character string to a long integer

51072 SYNOPSIS

```
51073     #include <wchar.h>
51074
51075     long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr,
51076                 int base);
51077     long long wcstoll(const wchar_t *restrict nptr,
51078                          wchar_t **restrict endptr, int base);
```

51078 DESCRIPTION

51079 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51080 conflict between the requirements described here and the ISO C standard is unintentional. This
51081 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

51082 These functions shall convert the initial portion of the wide-character string pointed to by *nptr* to
51083 **long**, **long long**, **unsigned long**, and **unsigned long long** representation, respectively. First, they
51084 shall decompose the input string into three parts:

- 51085 1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by
51086 *iswspace*())
- 51087 2. A subject sequence interpreted as an integer represented in some radix determined by the
51088 value of *base*
- 51089 3. A final wide-character string of one or more unrecognized wide-character codes, including
51090 the terminating null wide-character code of the input wide-character string

51091 Then they shall attempt to convert the subject sequence to an integer, and return the result.

51092 If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant,
51093 or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A decimal
51094 constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal
51095 constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7'
51096 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
51097 decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

51098 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
51099 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
51100 by a '+' or '-' sign, but not including an integer suffix. The letters from 'a' (or 'A') to 'z'
51101 (or 'z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less
51102 than that of *base* shall be permitted. If the value of *base* is 16, the wide-character code
51103 representations of 0x or 0X may optionally precede the sequence of letters and digits, following
51104 the sign if present.

51105 The subject sequence is defined as the longest initial subsequence of the input wide-character
51106 string, starting with the first non-white-space wide-character code that is of the expected form.
51107 The subject sequence contains no wide-character codes if the input wide-character string is
51108 empty or consists entirely of white-space wide-character code, or if the first non-white-space
51109 wide-character code is other than a sign or a permissible letter or digit.

51110 If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes
51111 starting with the first digit shall be interpreted as an integer constant. If the subject sequence has
51112 the expected form and the value of *base* is between 2 and 36, it shall be used as the base for
51113 conversion, ascribing to each letter its value as given above. If the subject sequence begins with a
51114 minus sign, the value resulting from the conversion shall be negated. A pointer to the final
51115 wide-character string shall be stored in the object pointed to by *endptr*, provided that *endptr* is

51116	not a null pointer.
51117 CX	In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.
51119	If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of <i>nptr</i> shall be stored in the object pointed to by <i>endptr</i> , provided that <i>endptr</i> is not a null pointer.
51122 CX	These functions shall not change the setting of <i>errno</i> if successful.
51123	Since 0, {LONG_MIN} or {LLONG_MIN} and {LONG_MAX} or {LLONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations
51124	should set <i>errno</i> to 0, then call <i>wcstol()</i> or <i>wcstoll()</i> , then check <i>errno</i> .
51126	RETURN VALUE
51127	Upon successful completion, these functions shall return the converted value, if any. If no
51128 CX	conversion could be performed, 0 shall be returned and <i>errno</i> may be set to indicate the error. If
51129	the correct value is outside the range of representable values, {LONG_MIN}, {LONG_MAX},
51130	{LLONG_MIN}, or {LLONG_MAX} shall be returned (according to the sign of the value), and
51131	<i>errno</i> set to [ERANGE].
51132	ERRORS
51133	These functions shall fail if:
51134 CX	[EINVAL] The value of <i>base</i> is not supported.
51135	[ERANGE] The value to be returned is not representable.
51136	These functions may fail if:
51137 CX	[EINVAL] No conversion could be performed.
51138	EXAMPLES
51139	None.
51140	APPLICATION USAGE
51141	None.
51142	RATIONALE
51143	None.
51144	FUTURE DIRECTIONS
51145	None.
51146	SEE ALSO
51147	<i>iswalph()</i> , <i>scanf()</i> , <i>wcstod()</i> , the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>
51148	CHANGE HISTORY
51149	First released in Issue 4. Derived from the MSE working draft.
51150	Issue 5
51151	The DESCRIPTION is updated to indicate that <i>errno</i> is not changed if the function is successful.
51152	Issue 6
51153	Extensions beyond the ISO C standard are marked.
51154	The following new requirements on POSIX implementations derive from alignment with the
51155	Single UNIX Specification:
51156	• In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
51157	added if no conversion could be performed.

- 51158 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:
- The `wcstol()` prototype is updated.
 - The `wcstoll()` function is added.
- 51159
- 51160

51161 NAME

51162 wcstold — convert a wide-character string to a double-precision number

51163 SYNOPSIS

51164 #include <wchar.h>
51165 long double wcstold(const wchar_t *restrict nptr,
51166 wchar_t **restrict endptr);

51167 DESCRIPTION

51168 Refer to *wcstod()*.

51169 NAME

51170 wcstoll — convert a wide-character string to a long integer

51171 SYNOPSIS

51172 #include <wchar.h>

51173 long long wcstoll(const wchar_t *restrict nptr,
51174 wchar_t **restrict endptr, int base);

51175 DESCRIPTION

51176 Refer to *wcstol()*.

51177 NAME

51178 `wcstombs` — convert a wide-character string to a character string

51179 SYNOPSIS

```
51180        #include <stdlib.h>
51181
51182        size_t wcstombs(char *restrict s, const wchar_t *restrict pwcs,
51183                size_t n);
```

51183 DESCRIPTION

51184 CX 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.

51187 The `wcstombs()` function shall convert the sequence of wide-character codes that are in the array pointed to by `pwcs` into a sequence of characters that begins in the initial shift state and store these characters into the array pointed to by `s`, stopping if a character would exceed the limit of `n` total bytes or if a null byte is stored. Each wide-character code shall be converted as if by a call to `wctomb()`, except that the shift state of `wctomb()` shall not be affected.

51192 The behavior of this function shall be affected by the `LC_CTYPE` category of the current locale.

51193 No more than `n` bytes shall be modified in the array pointed to by `s`. If copying takes place between objects that overlap, the behavior is undefined. If `s` is a null pointer, `wcstombs()` shall return the length required to convert the entire array regardless of the value of `n`, but no values are stored.

51197 The `wcstombs()` function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

51199 RETURN VALUE

51200 If a wide-character code is encountered that does not correspond to a valid character (of one or more bytes each), `wcstombs()` shall return (`size_t`)-1. Otherwise, `wcstombs()` shall return the number of bytes stored in the character array, not including any terminating null byte. The array shall not be null-terminated if the value returned is `n`.

51204 ERRORS

51205 The `wcstombs()` function may fail if:

51206 CX [EILSEQ] A wide-character code does not correspond to a valid character.

51207 EXAMPLES

51208 None.

51209 APPLICATION USAGE

51210 None.

51211 RATIONALE

51212 None.

51213 FUTURE DIRECTIONS

51214 None.

51215 SEE ALSO

51216 `mblen()`, `mbtowc()`, `mbstowcs()`, `wctomb()`, the Base Definitions volume of IEEE Std 1003.1-2001,
51217 <stdlib.h>

51218 CHANGE HISTORY

51219 First released in Issue 4. Derived from the ISO C standard.

51220 Issue 6

51221 The following new requirements on POSIX implementations derive from alignment with the
51222 Single UNIX Specification:

- 51223 • The DESCRIPTION states the effect of when *s* is a null pointer.
51224 • The [EILSEQ] error condition is added.

51225 The *wcstombs()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

51226 NAME

51227 wcstoul, wcstoull — convert a wide-character string to an unsigned long

51228 SYNOPSIS

```
51229     #include <wchar.h>
51230
51231     unsigned long wcstoul(const wchar_t *restrict nptr,
51232         wchar_t **restrict endptr, int base);
51233     unsigned long long wcstoull(const wchar_t *restrict nptr,
51234         wchar_t **restrict endptr, int base);
```

51234 DESCRIPTION

51235 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 51236 conflict between the requirements described here and the ISO C standard is unintentional. This
 51237 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

51238 The *wcstoul()* and *wcstoull()* functions shall convert the initial portion of the wide-character
 51239 string pointed to by *nptr* to **unsigned long** and **unsigned long long** representation, respectively.
 51240 First, they shall decompose the input wide-character string into three parts:

- 51241 1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by
 51242 *iswspace()*)
- 51243 2. A subject sequence interpreted as an integer represented in some radix determined by the
 51244 value of *base*
- 51245 3. A final wide-character string of one or more unrecognized wide-character codes, including
 51246 the terminating null wide-character code of the input wide-character string

51247 Then they shall attempt to convert the subject sequence to an unsigned integer, and return the
 51248 result.

51249 If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant,
 51250 or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A decimal
 51251 constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal
 51252 constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7'
 51253 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the
 51254 decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

51255 If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence
 51256 of letters and digits representing an integer with the radix specified by *base*, optionally preceded
 51257 by a '+' or '-' sign, but not including an integer suffix. The letters from 'a' (or 'A') to 'z'
 51258 (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less
 51259 than that of *base* shall be permitted. If the value of *base* is 16, the wide-character codes 0x or 0X
 51260 may optionally precede the sequence of letters and digits, following the sign if present.

51261 The subject sequence is defined as the longest initial subsequence of the input wide-character
 51262 string, starting with the first wide-character code that is not white space and is of the expected
 51263 form. The subject sequence contains no wide-character codes if the input wide-character string is
 51264 empty or consists entirely of white-space wide-character codes, or if the first wide-character
 51265 code that is not white space is other than a sign or a permissible letter or digit.

51266 If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes
 51267 starting with the first digit shall be interpreted as an integer constant. If the subject sequence has
 51268 the expected form and the value of *base* is between 2 and 36, it shall be used as the base for
 51269 conversion, ascribing to each letter its value as given above. If the subject sequence begins with a
 51270 minus sign, the value resulting from the conversion shall be negated. A pointer to the final
 51271 wide-character string shall be stored in the object pointed to by *endptr*, provided that *endptr* is

51272 not a null pointer.

51273 CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

51275 If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

51278 CX The *wcstoul()* function shall not change the setting of *errno* if successful.

51279 Since 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and 0 is also a valid return

51280 on success, an application wishing to check for error situations should set *errno* to 0, then call

51281 *wcstoul()* or *wcstoull()*, then check *errno*.

51282 RETURN VALUE

51283 Upon successful completion, the *wcstoul()* and *wcstoull()* functions shall return the converted

51284 CX value, if any. If no conversion could be performed, 0 shall be returned and *errno* may be set to

51285 indicate the error. If the correct value is outside the range of representable values,

51286 {ULONG_MAX} or {ULLONG_MAX} respectively shall be returned and *errno* set to [ERANGE].

51287 ERRORS

51288 These functions shall fail if:

51289 CX [EINVAL] The value of *base* is not supported.

51290 [ERANGE] The value to be returned is not representable.

51291 These functions may fail if:

51292 CX [EINVAL] No conversion could be performed.

51293 EXAMPLES

51294 None.

51295 APPLICATION USAGE

51296 None.

51297 RATIONALE

51298 None.

51299 FUTURE DIRECTIONS

51300 None.

51301 SEE ALSO

51302 *iswalpha()*, *scanf()*, *wcstod()*, *wcstol()*, the Base Definitions volume of IEEE Std 1003.1-2001,
51303 <wchar.h>

51304 CHANGE HISTORY

51305 First released in Issue 4. Derived from the MSE working draft.

51306 Issue 5

51307 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

51308 Issue 6

51309 Extensions beyond the ISO C standard are marked.

51310 The following new requirements on POSIX implementations derive from alignment with the
51311 Single UNIX Specification:

- The [EINVAL] error condition is added for when the value of *base* is not supported.

51313 In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
51314 added if no conversion could be performed.

51315 The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- 51316 • The `wcstoul()` prototype is updated.
51317 • The `wcstoull()` function is added.

51318 NAME

51319 wcstoumax — convert a wide-character string to an integer type

51320 SYNOPSIS

```
51321       #include <stddef.h>
51322       #include <inttypes.h>
51323       uintmax_t wcstoumax(const wchar_t *restrict nptr,
51324                        wchar_t **restrict endptr, int base);
```

51325 DESCRIPTION

51326 Refer to *wcstoiimax()*.

51327 NAME

51328 wcswcs — find a wide substring (**LEGACY**)

51329 SYNOPSIS

51330 XSI #include <wchar.h>

51331 wchar_t *wcs wcs(const wchar_t *ws1, const wchar_t *ws2);

51332

51333 DESCRIPTION

51334 The wcs wcs() function shall locate the first occurrence in the wide-character string pointed to by ws1 of the sequence of wide-character codes (excluding the terminating null wide-character code) in the wide-character string pointed to by ws2.

51337 RETURN VALUE

51338 Upon successful completion, wcs wcs() shall return a pointer to the located wide-character string or a null pointer if the wide-character string is not found.

51340 If ws2 points to a wide-character string with zero length, the function shall return ws1.

51341 ERRORS

51342 No errors are defined.

51343 EXAMPLES

51344 None.

51345 APPLICATION USAGE

51346 This function was not included in the final ISO/IEC 9899:1990/Amendment 1:1995 (E).
51347 Application developers are strongly encouraged to use the wcsstr() function instead.

51348 RATIONALE

51349 None.

51350 FUTURE DIRECTIONS

51351 This function may be withdrawn in a future version.

51352 SEE ALSO

51353 wcschr(), wcsstr(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

51354 CHANGE HISTORY

51355 First released in Issue 4. Derived from the MSE working draft.

51356 Issue 5

51357 Marked EX.

51358 Issue 6

51359 This function is marked LEGACY.

51360 NAME

51361 `wcswidth` — number of column positions of a wide-character string

51362 SYNOPSIS

51363 XSI `#include <wchar.h>`

51364 `int wcswidth(const wchar_t *pwcs, size_t n);`

51365

51366 DESCRIPTION

51367 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*.

51370 RETURN VALUE

51371 The `wcswidth()` function either shall return 0 (if *pwcs* points to a null wide-character code), or
51372 return the number of column positions to be occupied by the wide-character string pointed to by
51373 *pwcs*, or return -1 (if any of the first *n* wide-character codes in the wide-character string pointed
51374 to by *pwcs* is not a printable wide-character code).

51375 ERRORS

51376 No errors are defined.

51377 EXAMPLES

51378 None.

51379 APPLICATION USAGE

51380 This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the
51381 return value for a non-printable wide character is not specified.

51382 RATIONALE

51383 None.

51384 FUTURE DIRECTIONS

51385 None.

51386 SEE ALSO

51387 `wcwidth()`, the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.103, Column Position,
51388 `<wchar.h>`

51389 CHANGE HISTORY

51390 First released in Issue 4. Derived from the MSE working draft.

51391 Issue 6

51392 The Open Group Corrigendum U021/11 is applied. The function is marked as an extension.

51393 NAME

51394 wcsxfrm — wide-character string transformation

51395 SYNOPSIS

```
51396        #include <wchar.h>
51397        size_t wcsxfrm(wchar_t *restrict ws1, const wchar_t *restrict ws2,
51398                    size_t n);
```

51399 DESCRIPTION

51400 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51401 conflict between the requirements described here and the ISO C standard is unintentional. This
51402 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

51403 The *wcsxfrm()* function shall transform the wide-character string pointed to by *ws2* and place the
51404 resulting wide-character string into the array pointed to by *ws1*. The transformation shall be
51405 such that if *wcscmp()* is applied to two transformed wide strings, it shall return a value greater
51406 than, equal to, or less than 0, corresponding to the result of *wcscoll()* applied to the same two
51407 original wide-character strings. No more than *n* wide-character codes shall be placed into the
51408 resulting array pointed to by *ws1*, including the terminating null wide-character code. If *n* is 0,
51409 *ws1* is permitted to be a null pointer. If copying takes place between objects that overlap, the
51410 behavior is undefined.

51411 CX The *wcsxfrm()* function shall not change the setting of *errno* if successful.

51412 Since no return value is reserved to indicate an error, an application wishing to check for error
51413 situations should set *errno* to 0, then call *wcsxfrm()*, then check *errno*.

51414 RETURN VALUE

51415 The *wcsxfrm()* function shall return the length of the transformed wide-character string (not
51416 including the terminating null wide-character code). If the value returned is *n* or more, the
51417 contents of the array pointed to by *ws1* are unspecified.

51418 CX On error, the *wcsxfrm()* function may set *errno*, but no return value is reserved to indicate an
51419 error.

51420 ERRORS

51421 The *wcsxfrm()* function may fail if:

51422 CX [EINVAL] The wide-character string pointed to by *ws2* contains wide-character codes
51423 outside the domain of the collating sequence.

51424 EXAMPLES

51425 None.

51426 APPLICATION USAGE

51427 The transformation function is such that two transformed wide-character strings can be ordered
51428 by *wcscmp()* as appropriate to collating sequence information in the program's locale (category
51429 *LC_COLLATE*).

51430 The fact that when *n* is 0 *ws1* is permitted to be a null pointer is useful to determine the size of
51431 the *ws1* array prior to making the transformation.

51432 RATIONALE

51433 None.

51434 FUTURE DIRECTIONS

51435 None.

51436 SEE ALSO

51437 *wcscmp()*, *wcsncmp()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

51438 CHANGE HISTORY

51439 First released in Issue 4. Derived from the MSE working draft.

51440 Issue 5

51441 Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

51442 The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

51443 Issue 6

51444 In previous versions, this function was required to return -1 on error.

51445 Extensions beyond the ISO C standard are marked.

51446 The following new requirements on POSIX implementations derive from alignment with the
51447 Single UNIX Specification:

- 51448 • In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
51449 added if no conversion could be performed.

51450 The *wcsxfrm()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

51451 NAME

51452 `wctob` — wide-character to single-byte conversion

51453 SYNOPSIS

```
51454        #include <stdio.h>
51455        #include <wchar.h>
51456        int wctob(wint_t c);
```

51457 DESCRIPTION

51458 CX 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.

51461 The `wctob()` function shall determine whether *c* corresponds to a member of the extended character set whose character representation is a single byte when in the initial shift state.

51463 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

51464 RETURN VALUE

51465 The `wctob()` function shall return EOF if *c* does not correspond to a character with length one in the initial shift state. Otherwise, it shall return the single-byte representation of that character as an **unsigned char** converted to **int**.

51468 ERRORS

51469 No errors are defined.

51470 EXAMPLES

51471 None.

51472 APPLICATION USAGE

51473 None.

51474 RATIONALE

51475 None.

51476 FUTURE DIRECTIONS

51477 None.

51478 SEE ALSO

51479 `btowc()`, the Base Definitions volume of IEEE Std 1003.1-2001, <**wchar.h**>

51480 CHANGE HISTORY

51481 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
51482 (E).

51483 **NAME**

51484 wctomb — convert a wide-character code to a character

51485 **SYNOPSIS**

```
51486 #include <stdlib.h>
51487 int wctomb(char *s, wchar_t wchar);
```

51488 **DESCRIPTION**

51489 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51490 conflict between the requirements described here and the ISO C standard is unintentional. This
51491 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

51492 The *wctomb()* function shall determine the number of bytes needed to represent the character
51493 corresponding to the wide-character code whose value is *wchar* (including any change in the
51494 shift state). It shall store the character representation (possibly multiple bytes and any special
51495 bytes to change shift state) in the array object pointed to by *s* (if *s* is not a null pointer). At most
51496 {MB_CUR_MAX} bytes shall be stored. If *wchar* is 0, a null byte shall be stored, preceded by any
51497 shift sequence needed to restore the initial shift state, and *wctomb()* shall be left in the initial shift
51498 state.

51499 CX The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a
51500 state-dependent encoding, this function shall be placed into its initial state by a call for which its
51501 character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null
51502 pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a
51503 null pointer shall cause this function to return a non-zero value if encodings have state
51504 dependency, and 0 otherwise. Changing the *LC_CTYPE* category causes the shift state of this
51505 function to be unspecified.

51506 The *wctomb()* function need not be reentrant. A function that is not required to be reentrant is
51507 not required to be thread-safe.

51508 The implementation shall behave as if no function defined in this volume of
51509 IEEE Std 1003.1-2001 calls *wctomb()*.

51510 **RETURN VALUE**

51511 If *s* is a null pointer, *wctomb()* shall return a non-zero or 0 value, if character encodings,
51512 respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *wctomb()*
51513 shall return -1 if the value of *wchar* does not correspond to a valid character, or return the
51514 number of bytes that constitute the character corresponding to the value of *wchar*.

51515 In no case shall the value returned be greater than the value of the {MB_CUR_MAX} macro.

51516 **ERRORS**

51517 No errors are defined.

51518 **EXAMPLES**

51519 None.

51520 **APPLICATION USAGE**

51521 None.

51522 **RATIONALE**

51523 None.

51524 **FUTURE DIRECTIONS**

51525 None.

51526 SEE ALSO

51527 *mblen()*, *mbtowc()*, *mbstowcs()*, *wcstombs()*, the Base Definitions volume of IEEE Std 1003.1-2001,
51528 *<stdlib.h>*

51529 CHANGE HISTORY

51530 First released in Issue 4. Derived from the ANSI C standard.

51531 Issue 6

51532 Extensions beyond the ISO C standard are marked.

51533 In the DESCRIPTION, a note about reentrancy and thread-safety is added.

51534 NAME

51535 wctrans — define character mapping

51536 SYNOPSIS

```
51537 #include <wctype.h>
51538 wctrans_t wctrans(const char *charclass);
```

51539 DESCRIPTION

51540 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51541 conflict between the requirements described here and the ISO C standard is unintentional. This
51542 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

51543 The *wctrans()* function is defined for valid character mapping names identified in the current
51544 locale. The *charclass* is a string identifying a generic character mapping name for which codeset-
51545 specific information is required. The following character mapping names are defined in all
51546 locales: **tolower** and **toupper**.

51547 The function shall return a value of type **wctrans_t**, which can be used as the second argument
51548 to subsequent calls of *towctrans()*. The *wctrans()* function shall determine values of **wctrans_t**
51549 according to the rules of the coded character set defined by character mapping information in
51550 the program's locale (category *LC_CTYPE*). The values returned by *wctrans()* shall be valid until
51551 a call to *setlocale()* that modifies the category *LC_CTYPE*.

51552 RETURN VALUE

51553 CX The *wctrans()* function shall return 0 and may set *errno* to indicate the error if the given
51554 character mapping name is not valid for the current locale (category *LC_CTYPE*); otherwise, it
51555 shall return a non-zero object of type **wctrans_t** that can be used in calls to *towctrans()*.

51556 ERRORS

51557 The *wctrans()* function may fail if:

51558 CX [EINVAL] The character mapping name pointed to by *charclass* is not valid in the current
51559 locale.

51560 EXAMPLES

51561 None.

51562 APPLICATION USAGE

51563 None.

51564 RATIONALE

51565 None.

51566 FUTURE DIRECTIONS

51567 None.

51568 SEE ALSO

51569 *towctrans()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wctype.h>

51570 CHANGE HISTORY

51571 First released in Issue 5. Derived from ISO/IEC 9899:1990/Amendment 1:1995 (E).

51572 NAME

51573 `wctype` — define character class

51574 SYNOPSIS

```
51575        #include <wctype.h>
51576        wctype_t wctype(const char *property);
```

51577 DESCRIPTION

51578 CX 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.

51581 The `wctype()` function is defined for valid character class names as defined in the current locale. The `property` argument is a string identifying a generic character class for which codeset-specific type information is required. The following character class names shall be defined in all locales:

```
51584        alnum   digit   punct
51585        alpha   graph   space
51586        blank   lower   upper
51587        cntrl   print   xdigit
```

51588 Additional character class names defined in the locale definition file (category `LC_CTYPE`) can also be specified.

51590 The function shall return a value of type `wctype_t`, which can be used as the second argument to subsequent calls of `iswctype()`. The `wctype()` function shall determine values of `wctype_t` according to the rules of the coded character set defined by character type information in the program's locale (category `LC_CTYPE`). The values returned by `wctype()` shall be valid until a call to `setlocale()` that modifies the category `LC_CTYPE`.

51595 RETURN VALUE

51596 The `wctype()` function shall return 0 if the given character class name is not valid for the current locale (category `LC_CTYPE`); otherwise, it shall return an object of type `wctype_t` that can be used in calls to `iswctype()`.

51599 ERRORS

51600 No errors are defined.

51601 EXAMPLES

51602 None.

51603 APPLICATION USAGE

51604 None.

51605 RATIONALE

51606 None.

51607 FUTURE DIRECTIONS

51608 None.

51609 SEE ALSO

51610 `iswctype()`, the Base Definitions volume of IEEE Std 1003.1-2001, <wctype.h>

51611 CHANGE HISTORY

51612 First released in Issue 4.

51613 **Issue 5**

51614 The following change has been made in this issue for alignment with
51615 ISO/IEC 9899:1990/Amendment 1:1995 (E):

- 51616 • The SYNOPSIS has been changed to indicate that this function and associated data types are
51617 now made visible by inclusion of the <**wctype.h**> header rather than <**wchar.h**>.

51618 NAME

51619 `wcwidth` — number of column positions of a wide-character code

51620 SYNOPSIS

51621 XSI `#include <wchar.h>`

51622 `int wcwidth(wchar_t wc);`

51623

51624 DESCRIPTION

51625 The `wcwidth()` function shall determine the number of column positions required for the wide
51626 character `wc`. The application shall ensure that the value of `wc` is a character representable as a
51627 `wchar_t`, and is a wide-character code corresponding to a valid character in the current locale.

51628 RETURN VALUE

51629 The `wcwidth()` function shall either return 0 (if `wc` is a null wide-character code), or return the
51630 number of column positions to be occupied by the wide-character code `wc`, or return -1 (if `wc`
51631 does not correspond to a printable wide-character code).

51632 ERRORS

51633 No errors are defined.

51634 EXAMPLES

51635 None.

51636 APPLICATION USAGE

51637 This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the
51638 return value for a non-printable wide character is not specified.

51639 RATIONALE

51640 None.

51641 FUTURE DIRECTIONS

51642 None.

51643 SEE ALSO

51644 `wcswidth()`, the Base Definitions volume of IEEE Std 1003.1-2001, <`wchar.h`>

51645 CHANGE HISTORY

51646 First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
51647 draft.

51648 Issue 6

51649 The Open Group Corrigendum U021/12 is applied. This function is marked as an extension.

51650 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

51651 NAME

51652 wmemchr — find a wide character in memory

51653 SYNOPSIS

```
51654        #include <wchar.h>
51655        wchar_t *wmemchr(const wchar_t *ws, wchar_t wc, size_t n);
```

51656 DESCRIPTION

51657 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51658 conflict between the requirements described here and the ISO C standard is unintentional. This
51659 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

51660 The *wmemchr()* function shall locate the first occurrence of *wc* in the initial *n* wide characters of
51661 the object pointed to by *ws*. This function shall not be affected by locale and all **wchar_t** values
51662 shall be treated identically. The null wide character and **wchar_t** values not corresponding to
51663 valid characters shall not be treated specially.

51664 If *n* is zero, the application shall ensure that *ws* is a valid pointer and the function behaves as if
51665 no valid occurrence of *wc* is found.

51666 RETURN VALUE

51667 The *wmemchr()* function shall return a pointer to the located wide character, or a null pointer if
51668 the wide character does not occur in the object.

51669 ERRORS

51670 No errors are defined.

51671 EXAMPLES

51672 None.

51673 APPLICATION USAGE

51674 None.

51675 RATIONALE

51676 None.

51677 FUTURE DIRECTIONS

51678 None.

51679 SEE ALSO

51680 *wmemcmp()*, *wmemcpy()*, *wmemmove()*, *wmemset()*, the Base Definitions volume of
51681 IEEE Std 1003.1-2001, **<wchar.h>**

51682 CHANGE HISTORY

51683 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
51684 (E).

51685 Issue 6

51686 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

51687 NAME

51688 `wmemcmp — compare wide characters in memory`

51689 SYNOPSIS

```
51690        #include <wchar.h>
51691        int wmemcmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

51692 DESCRIPTION

51693 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51694 conflict between the requirements described here and the ISO C standard is unintentional. This
51695 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

51696 The `wmemcmp()` function shall compare the first *n* wide characters of the object pointed to by
51697 *ws1* to the first *n* wide characters of the object pointed to by *ws2*. This function shall not be
51698 affected by locale and all `wchar_t` values shall be treated identically. The null wide character and
51699 `wchar_t` values not corresponding to valid characters shall not be treated specially.

51700 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function
51701 shall behave as if the two objects compare equal.

51702 RETURN VALUE

51703 The `wmemcmp()` function shall return an integer greater than, equal to, or less than zero,
51704 respectively, as the object pointed to by *ws1* is greater than, equal to, or less than the object
51705 pointed to by *ws2*.

51706 ERRORS

51707 No errors are defined.

51708 EXAMPLES

51709 None.

51710 APPLICATION USAGE

51711 None.

51712 RATIONALE

51713 None.

51714 FUTURE DIRECTIONS

51715 None.

51716 SEE ALSO

51717 `wmemchr()`, `wmemcpy()`, `wmemmove()`, `wmemset()`, the Base Definitions volume of
51718 IEEE Std 1003.1-2001, `<wchar.h>`

51719 CHANGE HISTORY

51720 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
51721 (E).

51722 Issue 6

51723 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

51724 NAME

51725 wmemcpy — copy wide characters in memory

51726 SYNOPSIS

```
51727        #include <wchar.h>
51728        wchar_t *wmemcpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,
51729                    size_t n);
```

51730 DESCRIPTION

51731 CX The functionality described on this reference page is aligned with the ISO C standard. Any
51732 conflict between the requirements described here and the ISO C standard is unintentional. This
51733 volume of IEEE Std 1003.1-2001 defers to the ISO C standard.

51734 The *wmemcpy()* function shall copy *n* wide characters from the object pointed to by *ws2* to the
51735 object pointed to by *ws1*. This function shall not be affected by locale and all **wchar_t** values
51736 shall be treated identically. The null wide character and **wchar_t** values not corresponding to
51737 valid characters shall not be treated specially.

51738 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function
51739 shall copy zero wide characters.

51740 RETURN VALUE

51741 The *wmemcpy()* function shall return the value of *ws1*.

51742 ERRORS

51743 No errors are defined.

51744 EXAMPLES

51745 None.

51746 APPLICATION USAGE

51747 None.

51748 RATIONALE

51749 None.

51750 FUTURE DIRECTIONS

51751 None.

51752 SEE ALSO

51753 *wmemchr()*, *wmemcmp()*, *wmemmove()*, *wmemset()*, the Base Definitions volume of
51754 IEEE Std 1003.1-2001, <**wchar.h**>

51755 CHANGE HISTORY

51756 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
51757 (E).

51758 Issue 6

51759 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

51760 The *wmemcpy()* prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

51761 NAME

51762 **wmemmove** — copy wide characters in memory with overlapping areas

51763 SYNOPSIS

51764 `#include <wchar.h>`

51765 `wchar_t *wmemmove(wchar_t *ws1, const wchar_t *ws2, size_t n);`

51766 DESCRIPTION

51767 CX 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.

51770 The **wmemmove()** function shall copy *n* wide characters from the object pointed to by *ws2* to the object pointed to by *ws1*. Copying shall take place as if the *n* wide characters from the object pointed to by *ws2* are first copied into a temporary array of *n* wide characters that does not overlap the objects pointed to by *ws1* or *ws2*, and then the *n* wide characters from the temporary array are copied into the object pointed to by *ws1*.

51775 This function shall not be affected by locale and all **wchar_t** values shall be treated identically. The null wide character and **wchar_t** values not corresponding to valid characters shall not be treated specially.

51778 If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function shall copy zero wide characters.

51780 RETURN VALUE

51781 The **wmemmove()** function shall return the value of *ws1*.

51782 ERRORS

51783 No errors are defined

51784 EXAMPLES

51785 None.

51786 APPLICATION USAGE

51787 None.

51788 RATIONALE

51789 None.

51790 FUTURE DIRECTIONS

51791 None.

51792 SEE ALSO

51793 **wmemchr()**, **wmemcmp()**, **wmemcpy()**, **wmemset()**, the Base Definitions volume of
51794 IEEE Std 1003.1-2001, **<wchar.h>**

51795 CHANGE HISTORY

51796 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
51797 (E).

51798 Issue 6

51799 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

51800 NAME

51801 wmemset — set wide characters in memory

51802 SYNOPSIS

51803 #include <wchar.h>

51804 wchar_t *wmemset(wchar_t *ws, wchar_t wc, size_t n);

51805 DESCRIPTION

51806 CX 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.

51809 The *wmemset()* function shall copy the value of *wc* into each of the first *n* wide characters of the
51810 object pointed to by *ws*. This function shall not be affected by locale and all **wchar_t** values shall
51811 be treated identically. The null wide character and **wchar_t** values not corresponding to valid
51812 characters shall not be treated specially.

51813 If *n* is zero, the application shall ensure that *ws* is a valid pointer, and the function shall copy
51814 zero wide characters.

51815 RETURN VALUE

51816 The *wmemset()* functions shall return the value of *ws*.

51817 ERRORS

51818 No errors are defined.

51819 EXAMPLES

51820 None.

51821 APPLICATION USAGE

51822 None.

51823 RATIONALE

51824 None.

51825 FUTURE DIRECTIONS

51826 None.

51827 SEE ALSO

51828 *wmemchr()*, *wmemcmp()*, *wmemcpy()*, *wmemmove()*, the Base Definitions volume of
51829 IEEE Std 1003.1-2001, <wchar.h>

51830 CHANGE HISTORY

51831 First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995
51832 (E).

51833 Issue 6

51834 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

51835 NAME

51836 wordexp, wordfree — perform word expansions

51837 SYNOPSIS

```
51838 #include <wordexp.h>
51839 int wordexp(const char *restrict words, wordexp_t *restrict pwordexp,
51840           int flags);
51841 void wordfree(wordexp_t *pwordexp);
```

51842 DESCRIPTION

The *wordexp()* function shall perform word expansions as described in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.6, Word Expansions, subject to quoting as in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.2, Quoting, and place the list of expanded words into the structure pointed to by *pwordexp*.

The *words* argument is a pointer to a string containing one or more words to be expanded. The expansions shall be the same as would be performed by the command line interpreter if *words* were the part of a command line representing the arguments to a utility. Therefore, the application shall ensure that *words* does not contain an unquoted <newline> or any of the unquoted shell special characters '|', '&', ';', '<', '>' except in the context of command substitution as specified in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.6.3, Command Substitution. It also shall not contain unquoted parentheses or braces, except in the context of command or variable substitution. The application shall ensure that every member of *words* which it expects to have expanded by *wordexp()* does not contain an unquoted initial comment character. The application shall also ensure that any words which it intends to be ignored (because they begin or continue a comment) are deleted from *words*. If the argument *words* contains an unquoted comment character (number sign) that is the beginning of a token, *wordexp()* shall either treat the comment character as a regular character, or interpret it as a comment indicator and ignore the remainder of *words*.

The structure type **wordexp_t** is defined in the <wordexp.h> header and includes at least the following members:

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> .

The *wordexp()* function shall store the number of generated words into *pwordexp->we_wordc* and a pointer to a list of pointers to words in *pwordexp->we_wordv*. Each individual field created during field splitting (see the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.6.5, Field Splitting) or pathname expansion (see the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.6.6, Pathname Expansion) shall be a separate word in the *pwordexp->we_wordv* list. The words shall be in order as described in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.6, Word Expansions. The first pointer after the last word pointer shall be a null pointer. The expansion of special parameters described in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.5.2, Special Parameters is unspecified.

It is the caller's responsibility to allocate the storage pointed to by *pwordexp*. The *wordexp()* function shall allocate other space as needed, including memory pointed to by *pwordexp->we_wordv*. The *wordfree()* function frees any memory associated with *pwordexp* from a previous call to *wordexp()*.

51881 The *flags* argument is used to control the behavior of *wordexp()*. The value of *flags* is the
 51882 bitwise-inclusive OR of zero or more of the following constants, which are defined in
 51883 <wordexp.h>:

51884 WRDE_APPEND	Append words generated to the ones from a previous call to <i>wordexp()</i> .
51885 WRDE_DOOFFS	Make use of <i>pwordexp->we_offs</i> . If this flag is set, <i>pwordexp->we_offs</i> is used 51886 to specify how many null pointers to add to the beginning of 51887 <i>pwordexp->we_wordv</i> . In other words, <i>pwordexp->we_wordv</i> shall point to 51888 <i>pwordexp->we_offs</i> null pointers, followed by <i>pwordexp->we_wordc</i> word 51889 pointers, followed by a null pointer.
51890 WRDE_NOCMD	If the implementation supports the utilities defined in the Shell and 51891 Utilities volume of IEEE Std 1003.1-2001, fail if command substitution, as 51892 specified in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 51893 2.6.3, Command Substitution, is requested.
51894 WRDE_REUSE	The <i>pwordexp</i> argument was passed to a previous successful call to 51895 <i>wordexp()</i> , and has not been passed to <i>wordfree()</i> . The result shall be the 51896 same as if the application had called <i>wordfree()</i> and then called <i>wordexp()</i> 51897 without WRDE_REUSE.
51898 WRDE_SHOWERR	Do not redirect <i>stderr</i> to /dev/null.
51899 WRDE_UNDEF	Report error on an attempt to expand an undefined shell variable.

51900 The WRDE_APPEND flag can be used to append a new set of words to those generated by a
 51901 previous call to *wordexp()*. The following rules apply to applications when two or more calls to
 51902 *wordexp()* are made with the same value of *pwordexp* and without intervening calls to *wordfree()*:

- 51903 1. The first such call shall not set WRDE_APPEND. All subsequent calls shall set it.
- 51904 2. All of the calls shall set WRDE_DOOFFS, or all shall not set it.
- 51905 3. After the second and each subsequent call, *pwordexp->we_wordv* shall point to a list
 51906 containing the following:
 - 51907 a. Zero or more null pointers, as specified by WRDE_DOOFFS and *pwordexp->we_offs*
 - 51908 b. Pointers to the words that were in the *pwordexp->we_wordv* list before the call, in the
 51909 same order as before
 - 51910 c. Pointers to the new words generated by the latest call, in the specified order
- 51911 4. The count returned in *pwordexp->we_wordc* shall be the total number of words from all of
 51912 the calls.
- 51913 5. The application can change any of the fields after a call to *wordexp()*, but if it does it shall
 51914 reset them to the original value before a subsequent call, using the same *pwordexp* value, to
 51915 *wordfree()* or *wordexp()* with the WRDE_APPEND or WRDE_REUSE flag.

51916 If the implementation supports the utilities defined in the Shell and Utilities volume of
 51917 IEEE Std 1003.1-2001, and *words* contains an unquoted character—<newline>, '|', '&', ';',
 51918 '<', '>', '(', ')', '{', '}'—in an inappropriate context, *wordexp()* shall fail, and the number
 51919 of expanded words shall be 0.

51920 Unless WRDE_SHOWERR is set in *flags*, *wordexp()* shall redirect *stderr* to /dev/null for any
 51921 utilities executed as a result of command substitution while expanding *words*. If
 51922 WRDE_SHOWERR is set, *wordexp()* may write messages to *stderr* if syntax errors are detected
 51923 while expanding *words*.

51924 The application shall ensure that if WRDE_DOOFFS is set, then *pwordexp->we_offs* has the same
51925 value for each *wordexp()* call and *wordfree()* call using a given *pwordexp*.

51926 The following constants are defined as error return values:

51927 WRDE_BADCHAR	One of the unquoted characters—<newline>, ' ', '&', ';', '<', '>', 51928 '(' , ')' , '{' , '}' —appears in <i>words</i> in an inappropriate context.
51929 WRDE_BADVAL	Reference to undefined shell variable when WRDE_UNDEF is set in <i>flags</i> .
51930 WRDE_CMDSUB	Command substitution requested when WRDE_NOCMD was set in <i>flags</i> .
51931 WRDE_NOSPACE	Attempt to allocate memory failed.
51932 WRDE_SYNTAX	Shell syntax error, such as unbalanced parentheses or unterminated string.

51934 **RETURN VALUE**

51935 Upon successful completion, *wordexp()* shall return 0. Otherwise, a non-zero value, as described
51936 in <*wordexp.h*>, shall be returned to indicate an error. If *wordexp()* returns the value
51937 WRDE_NOSPACE, then *pwordexp->we_wordc* and *pwordexp->we_wordv* shall be updated to
51938 reflect any words that were successfully expanded. In other cases, they shall not be modified.

51939 The *wordfree()* function shall not return a value.

51940 **ERRORS**

51941 No errors are defined.

51942 **EXAMPLES**

51943 None.

51944 **APPLICATION USAGE**

51945 The *wordexp()* function is intended to be used by an application that wants to do all of the shell's
51946 expansions on a word or words obtained from a user. For example, if the application prompts
51947 for a filename (or list of filenames) and then uses *wordexp()* to process the input, the user could
51948 respond with anything that would be valid as input to the shell.

51949 The WRDE_NOCMD flag is provided for applications that, for security or other reasons, want to
51950 prevent a user from executing shell commands. Disallowing unquoted shell special characters
51951 also prevents unwanted side effects, such as executing a command or writing a file.

51952 **RATIONALE**

51953 This function was included as an alternative to *glob()*. There had been continuing controversy
51954 over exactly what features should be included in *glob()*. It is hoped that by providing *wordexp()*
51955 (which provides all of the shell word expansions, but which may be slow to execute) and *glob()*
51956 (which is faster, but which only performs pathname expansion, without tilde or parameter
51957 expansion) this will satisfy the majority of applications.

51958 While *wordexp()* could be implemented entirely as a library routine, it is expected that most
51959 implementations run a shell in a subprocess to do the expansion.

51960 Two different approaches have been proposed for how the required information might be
51961 presented to the shell and the results returned. They are presented here as examples.

51962 One proposal is to extend the *echo* utility by adding a -q option. This option would cause *echo* to
51963 add a backslash before each backslash and <blank> that occurs within an argument. The
51964 *wordexp()* function could then invoke the shell as follows:

```
51965 (void) strcpy(buffer, "echo -q");
51966 (void) strcat(buffer, words);
51967 if ((flags & WRDE_SHOWERR) == 0)
```

```
51968     (void) strcat(buffer, "2>/dev/null");
51969     f = popen(buffer, "r");
```

51970 The *wordexp()* function would read the resulting output, remove unquoted backslashes, and
51971 break into words at unquoted <blank>s. If the WRDE_NOCMD flag was set, *wordexp()* would
51972 have to scan *words* before starting the subshell to make sure that there would be no command
51973 substitution. In any case, it would have to scan *words* for unquoted special characters.

51974 Another proposal is to add the following options to *sh*:

51975 **-w wordlist**

51976 This option provides a wordlist expansion service to applications. The words in *wordlist*
51977 shall be expanded and the following written to standard output:

- 51978 1. The count of the number of words after expansion, in decimal, followed by a null byte
- 51979 2. The number of bytes needed to represent the expanded words (not including null
51980 separators), in decimal, followed by a null byte
- 51981 3. The expanded words, each terminated by a null byte

51982 If an error is encountered during word expansion, *sh* exits with a non-zero status after
51983 writing the former to report any words successfully expanded

51984 **-P** Run in “protected” mode. If specified with the **-w** option, no command substitution shall
51985 be performed.

51986 With these options, *wordexp()* could be implemented fairly simply by creating a subprocess
51987 using *fork()* and executing *sh* using the line:

```
51988 execl(<shell path>, "sh", "-P", "-w", words, (char *)0);
```

51989 after directing standard error to **/dev/null**.

51990 It seemed objectionable for a library routine to write messages to standard error, unless
51991 explicitly requested, so *wordexp()* is required to redirect standard error to **/dev/null** to ensure
51992 that no messages are generated, even for commands executed for command substitution. The
51993 WRDE_SHOWERR flag can be specified to request that error messages be written.

51994 The WRDE_REUSE flag allows the implementation to avoid the expense of freeing and
51995 reallocating memory, if that is possible. A minimal implementation can call *wordfree()* when
51996 WRDE_REUSE is set.

51997 FUTURE DIRECTIONS

51998 None.

51999 SEE ALSO

52000 *fnmatch()*, *glob()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<wordexp.h>**, the Shell
52001 and Utilities volume of IEEE Std 1003.1-2001, Chapter 2, Shell Command Language

52002 CHANGE HISTORY

52003 First released in Issue 4. Derived from the ISO POSIX-2 standard.

52004 Issue 5

52005 Moved from POSIX2 C-language Binding to BASE.

52006 Issue 6

52007 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

52008 The **restrict** keyword is added to the *wordexp()* prototype for alignment with the
52009 ISO/IEC 9899:1999 standard.

52010 NAME

52011 wprintf — print formatted wide-character output

52012 SYNOPSIS

52013 #include <stdio.h>
52014 #include <wchar.h>

52015 int wprintf(const wchar_t *restrict *format*, ...);

52016 DESCRIPTION

52017 Refer to *fwprintf()*.

52018 NAME

52019 pwrite, write — write on a file

52020 SYNOPSIS

```
52021     #include <unistd.h>
52022 XSI     ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
52023             off_t offset);
52024     ssize_t write(int fildes, const void *buf, size_t nbyte);
```

52025 DESCRIPTION

52026 The *write()* function shall attempt to write *nbyte* bytes from the buffer pointed to by *buf* to the
52027 file associated with the open file descriptor, *fildes*.

52028 Before any action described below is taken, and if *nbyte* is zero and the file is a regular file, the
52029 *write()* function may detect and return errors as described below. In the absence of errors, or if
52030 error detection is not performed, the *write()* function shall return zero and have no other results.
52031 If *nbyte* is zero and the file is not a regular file, the results are unspecified.

52032 On a regular file or other file capable of seeking, the actual writing of data shall proceed from the
52033 position in the file indicated by the file offset associated with *fildes*. Before successful return
52034 from *write()*, the file offset shall be incremented by the number of bytes actually written. On a
52035 regular file, if this incremented file offset is greater than the length of the file, the length of the
52036 file shall be set to this file offset.

52037 On a file not capable of seeking, writing shall always take place starting at the current position.
52038 The value of a file offset associated with such a device is undefined.

52039 If the O_APPEND flag of the file status flags is set, the file offset shall be set to the end of the file
52040 prior to each write and no intervening file modification operation shall occur between changing
52041 the file offset and the write operation.

52042 XSI If a *write()* requests that more bytes be written than there is room for (for example, the process'
52043 file size limit or the physical end of a medium), only as many bytes as there is room for shall be
52044 written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A
52045 write of 512 bytes will return 20. The next write of a non-zero number of bytes would give a
52046 failure return (except as noted below).

52047 XSI If the request would cause the file size to exceed the soft file size limit for the process and there
52048 is no room for any bytes to be written, the request shall fail and the implementation shall
52049 generate the SIGXFSZ signal for the thread.

52050 If *write()* is interrupted by a signal before it writes any data, it shall return -1 with *errno* set to
52051 [EINTR].

52052 If *write()* is interrupted by a signal after it successfully writes some data, it shall return the
52053 number of bytes written.

52054 If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-defined.

52055 After a *write()* to a regular file has successfully returned:

- 52056 • Any successful *read()* from each byte position in the file that was modified by that write shall
52057 return the data specified by the *write()* for that position until such byte positions are again
52058 modified.
- 52059 • Any subsequent successful *write()* to the same byte position in the file shall overwrite that
52060 file data.

52061 Write requests to a pipe or FIFO shall be handled in the same way as a regular file with the
52062 following exceptions:

- 52063 • There is no file offset associated with a pipe, hence each write request shall append to the
52064 end of the pipe.
- 52065 • Write requests of {PIPE_BUF} bytes or less shall not be interleaved with data from other
52066 processes doing writes on the same pipe. Writes of greater than {PIPE_BUF} bytes may have
52067 data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the
52068 O_NONBLOCK flag of the file status flags is set.
- 52069 • If the O_NONBLOCK flag is clear, a write request may cause the thread to block, but on
52070 normal completion it shall return *nbyte*.
- 52071 • If the O_NONBLOCK flag is set, *write()* requests shall be handled differently, in the
52072 following ways:
 - 52073 — The *write()* function shall not block the thread.
 - 52074 — A write request for {PIPE_BUF} or fewer bytes shall have the following effect: if there is
52075 sufficient space available in the pipe, *write()* shall transfer all the data and return the
52076 number of bytes requested. Otherwise, *write()* shall transfer no data and return -1 with
52077 *errno* set to [EAGAIN].
 - 52078 — A write request for more than {PIPE_BUF} bytes shall cause one of the following:
 - 52079 — When at least one byte can be written, transfer what it can and return the number of
52080 bytes written. When all data previously written to the pipe is read, it shall transfer at
52081 least {PIPE_BUF} bytes.
 - 52082 — When no data can be written, transfer no data, and return -1 with *errno* set to
52083 [EAGAIN].

52084 When attempting to write to a file descriptor (other than a pipe or FIFO) that supports non-
52085 blocking writes and cannot accept the data immediately:

- 52086 • If the O_NONBLOCK flag is clear, *write()* shall block the calling thread until the data can be
52087 accepted.
- 52088 • If the O_NONBLOCK flag is set, *write()* shall not block the thread. If some data can be
52089 written without blocking the thread, *write()* shall write what it can and return the number of
52090 bytes written. Otherwise, it shall return -1 and set *errno* to [EAGAIN].

52091 Upon successful completion, where *nbyte* is greater than 0, *write()* shall mark for update the
52092 *st_ctime* and *st_mtime* fields of the file, and if the file is a regular file, the S_ISUID and S_ISGID
52093 bits of the file mode may be cleared.

52094 For regular files, no data transfer shall occur past the offset maximum established in the open
52095 file description associated with *fd*.

52096 If *fd* refers to a socket, *write()* shall be equivalent to *send()* with no flags set.

52097 SIO If the O_DSYNC bit has been set, write I/O operations on the file descriptor shall complete as
52098 defined by synchronized I/O data integrity completion.

52099 If the O_SYNC bit has been set, write I/O operations on the file descriptor shall complete as
52100 defined by synchronized I/O file integrity completion.

52101 SHM If *fd* refers to a shared memory object, the result of the *write()* function is unspecified.

52102 TTY If *fd* refers to a typed memory object, the result of the *write()* function is unspecified.

52103 XSR If *fildes* refers to a STREAM, the operation of *write()* shall be determined by the values of the minimum and maximum *nbyte* range (packet size) accepted by the STREAM. These values are determined by the topmost STREAM module. If *nbyte* falls within the packet size range, *nbyte* bytes shall be written. If *nbyte* does not fall within the range and the minimum packet size value is 0, *write()* shall break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, *write()* shall fail with *errno* set to [ERANGE]. Writing a zero-length buffer (*nbyte* is 0) to a STREAMS device sends 0 bytes with 0 returned. However, writing a zero-length buffer to a STREAMS-based pipe or FIFO sends no message and 0 is returned. The process may issue I_SWROPT *ioctl()* to enable zero-length messages to be sent across the pipe or FIFO.

52114 When writing to a STREAM, data messages are created with a priority band of 0. When writing
52115 to a STREAM that is not a pipe or FIFO:

- If O_NONBLOCK is clear, and the STREAM cannot accept data (the STREAM write queue is full due to internal flow control conditions), *write()* shall block until data can be accepted.
- If O_NONBLOCK is set and the STREAM cannot accept data, *write()* shall return -1 and set *errno* to [EAGAIN].
- If O_NONBLOCK is set and part of the buffer has been written while a condition in which the STREAM cannot accept additional data occurs, *write()* shall terminate and return the number of bytes written.

52123 In addition, *write()* shall fail if the STREAM head has processed an asynchronous error before
52124 the call. In this case, the value of *errno* does not reflect the result of *write()*, but reflects the prior
52125 error.

52126 XSI The *pwrite()* function shall be equivalent to *write()*, except that it writes into a given position without changing the file pointer. The first three arguments to *pwrite()* are the same as *write()* with the addition of a fourth argument offset for the desired position inside the file.

52129 RETURN VALUE

52130 XSI Upon successful completion, *write()* and *pwrite()* shall return the number of bytes actually written to the file associated with *fildes*. This number shall never be greater than *nbyte*. Otherwise, -1 shall be returned and *errno* set to indicate the error.

52133 ERRORS

52134 XSI The *write()* and *pwrite()* functions shall fail if:

52135 [EAGAIN]	The O_NONBLOCK flag is set for the file descriptor and the thread would be delayed in the <i>write()</i> operation.
52137 [EBADF]	The <i>fildes</i> argument is not a valid file descriptor open for writing.
52138 [EFBIG]	An attempt was made to write a file that exceeds the implementation-defined maximum file size or the process' file size limit, and there was no room for any bytes to be written.
52141 [EFBIG]	The file is a regular file, <i>nbyte</i> is greater than 0, and the starting position is greater than or equal to the offset maximum established in the open file description associated with <i>fildes</i> .
52144 [EINTR]	The write operation was terminated due to the receipt of a signal, and no data was transferred.
52146 [EIO]	The process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor

52148		blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.
52149		
52150	[ENOSPC]	There was no free space remaining on the device containing the file.
52151	[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.
52152		
52153		
52154 XSR	[ERANGE]	The transfer request size was outside the range supported by the STREAMS file associated with <i>fildes</i> .
52155		
52156		The <i>write()</i> function shall fail if:
52157	[EAGAIN] or [EWOULDBLOCK]	
52158		The file descriptor is for a socket, is marked O_NONBLOCK, and write would block.
52159		
52160	[ECONNRESET]	A write was attempted on a socket that is not connected.
52161	[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, a SIGPIPE signal shall also be sent to the thread.
52162		
52163		
52164 XSI		The <i>write()</i> and <i>pwrite()</i> functions may fail if:
52165 XSR	[EINVAL]	The STREAM or multiplexer referenced by <i>fildes</i> is linked (directly or indirectly) downstream from a multiplexer.
52166		
52167	[EIO]	A physical I/O error has occurred.
52168	[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
52169	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
52170		
52171 XSR	[ENXIO]	A hangup occurred on the STREAM being written to.
52172 XSR		A write to a STREAMS file may fail if an error message has been received at the STREAM head.
52173		In this case, <i>errno</i> is set to the value included in the error message.
52174		The <i>write()</i> function may fail if:
52175	[EACCES]	A write was attempted on a socket and the calling process does not have appropriate privileges.
52176		
52177	[ENETDOWN]	A write was attempted on a socket and the local network interface used to reach the destination is down.
52178		
52179	[ENETUNREACH]	
52180		A write was attempted on a socket and no route to the network is present.
52181 XSI		The <i>pwrite()</i> function shall fail and the file pointer remain unchanged if:
52182 XSI	[EINVAL]	The <i>offset</i> argument is invalid. The value is negative.
52183 XSI	[ESPIPE]	<i>fildes</i> is associated with a pipe or FIFO.

2
2

52184 EXAMPLES

52185 Writing from a Buffer

52186 The following example writes data from the buffer pointed to by *buf* to the file associated with
 52187 the file descriptor *fd*.

```
52188 #include <sys/types.h>
52189 #include <string.h>
52190 ...
52191 char buf[20];
52192 size_t nbytes;
52193 ssize_t bytes_written;
52194 int fd;
52195 ...
52196 strcpy(buf, "This is a test\n");
52197 nbytes = strlen(buf);
52198 bytes_written = write(fd, buf, nbytes);
52199 ...
```

52200 APPLICATION USAGE

52201 None.

52202 RATIONALE

52203 See also the RATIONALE section in *read()*.

52204 An attempt to write to a pipe or FIFO has several major characteristics:

- *Atomic/non-atomic*: A write is atomic if the whole amount written in one operation is not interleaved with data from any other process. This is useful when there are multiple writers sending data to a single reader. Applications need to know how large a write request can be expected to be performed atomically. This maximum is called {PIPE_BUF}. This volume of IEEE Std 1003.1-2001 does not say whether write requests for more than {PIPE_BUF} bytes are atomic, but requires that writes of {PIPE_BUF} or fewer bytes shall be atomic.

- *Blocking/immediate*: Blocking is only possible with O_NONBLOCK clear. If there is enough space for all the data requested to be written immediately, the implementation should do so. Otherwise, the calling thread may block; that is, pause until enough space is available for writing. The effective size of a pipe or FIFO (the maximum amount that can be written in one operation without blocking) may vary dynamically, depending on the implementation, so it is not possible to specify a fixed value for it.

52217 • *Complete/partial/deferred*: A write request:

```
52218     int fildes;
52219     size_t nbyte;
52220     ssize_t ret;
52221     char *buf;
52222
52223     ret = write(fildes, buf, nbyte);
```

52224 may return:

52224 Complete	<i>ret=nbyte</i>
52225 Partial	<i>ret<nbyte</i>

52226 This shall never happen if *nbyte*≤{PIPE_BUF}. If it does happen (with
 52227 *nbyte*>{PIPE_BUF}), this volume of IEEE Std 1003.1-2001 does not guarantee

52228 atomicity, even if $ret \leq \{\text{PIPE_BUF}\}$, because atomicity is guaranteed according
 52229 to the amount requested, not the amount written.

52230 Deferred: $ret = -1$, $errno = [\text{EAGAIN}]$

52231 This error indicates that a later request may succeed. It does not indicate that it
 52232 shall succeed, even if $nbyte \leq \{\text{PIPE_BUF}\}$, because if no process reads from the
 52233 pipe or FIFO, the write never succeeds. An application could usefully count the
 52234 number of times [EAGAIN] is caused by a particular value of
 52235 $nbyte > \{\text{PIPE_BUF}\}$ and perhaps do later writes with a smaller value, on the
 52236 assumption that the effective size of the pipe may have decreased.

52237 Partial and deferred writes are only possible with O_NONBLOCK set.

52238 The relations of these properties are shown in the following tables:

Write to a Pipe or FIFO with O_NONBLOCK clear			
Immediately Writable:	None	Some	$nbyte$
$nbyte \leq \{\text{PIPE_BUF}\}$	Atomic blocking $nbyte$	Atomic blocking $nbyte$	Atomic immediate $nbyte$
$nbyte > \{\text{PIPE_BUF}\}$	Blocking $nbyte$	Blocking $nbyte$	Blocking $nbyte$

52245 If the O_NONBLOCK flag is clear, a write request shall block if the amount writable
 52246 immediately is less than that requested. If the flag is set (by *fcntl()*), a write request shall never
 52247 block.

Write to a Pipe or FIFO with O_NONBLOCK set			
Immediately Writable:	None	Some	$nbyte$
$nbyte \leq \{\text{PIPE_BUF}\}$	-1, [EAGAIN]	-1, [EAGAIN]	Atomic $nbyte$
$nbyte > \{\text{PIPE_BUF}\}$	-1, [EAGAIN]	< $nbyte$ or -1, [EAGAIN]	$\leq nbyte$ or -1, [EAGAIN]

52254 There is no exception regarding partial writes when O_NONBLOCK is set. With the exception
 52255 of writing to an empty pipe, this volume of IEEE Std 1003.1-2001 does not specify exactly when a
 52256 partial write is performed since that would require specifying internal details of the
 52257 implementation. Every application should be prepared to handle partial writes when
 52258 O_NONBLOCK is set and the requested amount is greater than {PIPE_BUF}, just as every
 52259 application should be prepared to handle partial writes on other kinds of file descriptors.

52260 The intent of forcing writing at least one byte if any can be written is to assure that each write
 52261 makes progress if there is any room in the pipe. If the pipe is empty, {PIPE_BUF} bytes must be
 52262 written; if not, at least some progress must have been made.

52263 Where this volume of IEEE Std 1003.1-2001 requires -1 to be returned and *errno* set to
 52264 [EAGAIN], most historical implementations return zero (with the O_NDELAY flag set, which is
 52265 the historical predecessor of O_NONBLOCK, but is not itself in this volume of
 52266 IEEE Std 1003.1-2001). The error indications in this volume of IEEE Std 1003.1-2001 were chosen
 52267 so that an application can distinguish these cases from end-of-file. While *write()* cannot receive
 52268 an indication of end-of-file, *read()* can, and the two functions have similar return values. Also,
 52269 some existing systems (for example, Eighth Edition) permit a write of zero bytes to mean that
 52270 the reader should get an end-of-file indication; for those systems, a return value of zero from
 52271 *write()* indicates a successful write of an end-of-file indication.

52272 Implementations are allowed, but not required, to perform error checking for *write()* requests of
52273 zero bytes.

52274 The concept of a {PIPE_MAX} limit (indicating the maximum number of bytes that can be
52275 written to a pipe in a single operation) was considered, but rejected, because this concept would
52276 unnecessarily limit application writing.

52277 See also the discussion of O_NONBLOCK in *read()*.

52278 Writes can be serialized with respect to other reads and writes. If a *read()* of file data can be
52279 proven (by any means) to occur after a *write()* of the data, it must reflect that *write()*, even if the
52280 calls are made by different processes. A similar requirement applies to multiple write operations
52281 to the same file position. This is needed to guarantee the propagation of data from *write()* calls
52282 to subsequent *read()* calls. This requirement is particularly significant for networked file
52283 systems, where some caching schemes violate these semantics.

52284 Note that this is specified in terms of *read()* and *write()*. The XSI extensions *readv()* and *writev()*
52285 also obey these semantics. A new “high-performance” write analog that did not follow these
52286 serialization requirements would also be permitted by this wording. This volume of
52287 IEEE Std 1003.1-2001 is also silent about any effects of application-level caching (such as that
52288 done by *stdio*).

52289 This volume of IEEE Std 1003.1-2001 does not specify the value of the file offset after an error is
52290 returned; there are too many cases. For programming errors, such as [EBADF], the concept is
52291 meaningless since no file is involved. For errors that are detected immediately, such as
52292 [EAGAIN], clearly the pointer should not change. After an interrupt or hardware error, however,
52293 an updated value would be very useful and is the behavior of many implementations.

52294 This volume of IEEE Std 1003.1-2001 does not specify behavior of concurrent writes to a file from
52295 multiple processes. Applications should use some form of concurrency control.

52296 FUTURE DIRECTIONS

52297 None.

52298 SEE ALSO

52299 *chmod()*, *creat()*, *dup()*, *fcntl()*, *getrlimit()*, *lseek()*, *open()*, *pipe()*, *ulimit()*, *writev()*, the Base
52300 Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <stropts.h>, <sys/uio.h>, <unistd.h>

52301 CHANGE HISTORY

52302 First released in Issue 1. Derived from Issue 1 of the SVID.

52303 Issue 5

52304 The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX
52305 Threads Extension.

52306 Large File Summit extensions are added.

52307 The *pwrite()* function is added.

52308 Issue 6

52309 The DESCRIPTION states that the *write()* function does not block the thread. Previously this
52310 said “process” rather than “thread”.

52311 The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are
52312 marked as part of the XSI STREAMS Option Group.

52313 The following new requirements on POSIX implementations derive from alignment with the
52314 Single UNIX Specification:

- 52315 • The DESCRIPTION now states that if *write()* is interrupted by a signal after it has
52316 successfully written some data, it returns the number of bytes written. In the POSIX.1-1988
52317 standard, it was optional whether *write()* returned the number of bytes written, or whether it
52318 returned -1 with *errno* set to [EINTR]. This is a FIPS requirement.

- 52319 • The following changes are made to support large files:

- 52320 — For regular files, no data transfer occurs past the offset maximum established in the open
52321 file description associated with the *fd*.
52322 — A second [EFBIG] error condition is added.

- 52323 • The [EIO] error condition is added.

- 52324 • The [EPIPE] error condition is added for when a pipe has only one end open.

- 52325 • The [ENXIO] optional error condition is added.

52326 Text referring to sockets is added to the DESCRIPTION.

52327 The following changes were made to align with the IEEE P1003.1a draft standard:

- 52328 • The effect of reading zero bytes is clarified.

52329 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
52330 *write()* results are unspecified for typed memory objects.

52331 The following error conditions are added for operations on sockets: [EAGAIN],
52332 [EWOULDBLOCK], [ECONNRESET], [ENOTCONN], and [EPIPE].

52333 The [EIO] error is made optional. 1

52334 The [ENOBUFS] error is added for sockets.

52335 The following error conditions are added for operations on sockets: [EACCES], [ENETDOWN],
52336 and [ENETUNREACH].

52337 The *writev()* function is split out into a separate reference page.

52338 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/146 is applied, updating text in the 2
52339 ERRORS section from "a SIGPIPE signal is generated to the calling process" to "a SIGPIPE 2
52340 signal shall also be sent to the thread". 2

52341 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/147 is applied, making a correction to the 2
52342 RATIONALE. 2

52343 NAME

52344 writev — write a vector

52345 SYNOPSIS

52346 XSI #include <sys/uio.h>

```
52347 ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

52348

52349 DESCRIPTION

52350 The *writev()* function shall be equivalent to *write()*, except as described below. The *writev()* function shall gather output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov[0]*, *iov[1]*, ..., *iov[iovcnt-1]*. The *iovcnt* argument is valid if greater than 0 and less than or equal to {IOV_MAX}, as defined in <limits.h>.

52354 Each *iovec* entry specifies the base address and length of an area in memory from which data
52355 should be written. The *writev()* function shall always write a complete area before proceeding to
52356 the next.

52357 If *fildes* refers to a regular file and all of the *iov_len* members in the array pointed to by *iov* are 0,
52358 *writev()* shall return 0 and have no other effect. For other file types, the behavior is unspecified.

52359 If the sum of the *iov_len* values is greater than {SSIZE_MAX}, the operation shall fail and no data
52360 shall be transferred.

52361 RETURN VALUE

52362 Upon successful completion, *writev()* shall return the number of bytes actually written.
52363 Otherwise, it shall return a value of -1, the file-pointer shall remain unchanged, and *errno* shall
52364 be set to indicate an error.

52365 ERRORS

52366 Refer to *write()*.

52367 In addition, the *writev()* function shall fail if:

52368 [EINVAL] The sum of the *iov_len* values in the *iov* array would overflow an *ssize_t*.

52369 The *writev()* function may fail and set *errno* to:

52370 [EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.

52371 EXAMPLES

52372 Writing Data from an Array

52373 The following example writes data from the buffers specified by members of the *iov* array to the
52374 file associated with the file descriptor *fd*.

```
52375 #include <sys/types.h>
52376 #include <sys/uio.h>
52377 #include <unistd.h>
52378 ...
52379 ssize_t bytes_written;
52380 int fd;
52381 char *buf0 = "short string\n";
52382 char *buf1 = "This is a longer string\n";
52383 char *buf2 = "This is the longest string in this example\n";
52384 int iovcnt;
52385 struct iovec iov[3];
```

```
52386     iov[0].iov_base = buf0;
52387     iov[0].iov_len = strlen(buf0);
52388     iov[1].iov_base = buf1;
52389     iov[1].iov_len = strlen(buf1);
52390     iov[2].iov_base = buf2;
52391     iov[2].iov_len = strlen(buf2);
52392     ...
52393     iovcnt = sizeof(iov) / sizeof(struct iovec);
52394     bytes_written = writev(fd, iov, iovcnt);
52395     ...
```

52396 APPLICATION USAGE

52397 None.

52398 RATIONALE

52399 Refer to *write()*.

52400 FUTURE DIRECTIONS

52401 None.

52402 SEE ALSO

52403 *readv()*, *write()*, the Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <sys/uio.h>

52404 CHANGE HISTORY

52405 First released in Issue 4, Version 2.

52406 Issue 6

52407 Split out from the *write()* reference page.

52408 NAME

52409 **wscanf** — convert formatted wide-character input

52410 SYNOPSIS

```
52411        #include <stdio.h>
52412        #include <wchar.h>
52413        int wscanf(const wchar_t *restrict format, ... );
```

52414 DESCRIPTION

52415 Refer to *fwscanf()*.

52416 NAME

52417 *y0, y1, yn* — Bessel functions of the second kind

52418 SYNOPSIS

```
52419 XSI #include <math.h>
52420     double y0(double x);
52421     double y1(double x);
52422     double yn(int n, double x);
```

52424 DESCRIPTION

52425 The *y0()*, *y1()*, and *yn()* functions shall compute Bessel functions of *x* of the second kind of
52426 orders 0, 1, and *n*, respectively.

52427 An application wishing to check for error situations should set *errno* to zero and call
52428 *feclearexcept(FE_ALL_EXCEPT)* before calling these functions. On return, if *errno* is non-zero or
52429 *fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)* is non-
52430 zero, an error has occurred.

52431 RETURN VALUE

52432 Upon successful completion, these functions shall return the relevant Bessel value of *x* of the
52433 second kind.

52434 If *x* is NaN, NaN shall be returned.

52435 If the *x* argument to these functions is negative, -HUGE_VAL or NaN shall be returned, and a
52436 domain error may occur.

52437 If *x* is 0.0, -HUGE_VAL shall be returned and a pole error may occur. 2

52438 If the correct result would cause underflow, 0.0 shall be returned and a range error may occur.

52439 If the correct result would cause overflow, -HUGE_VAL or 0.0 shall be returned and a range
52440 error may occur.

52441 ERRORS

52442 These functions may fail if:

52443 Domain Error The value of *x* is negative.

52444 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
52445 then *errno* shall be set to [EDOM]. If the integer expression (*math_errhandling*
52446 & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception
52447 shall be raised.

52448 Pole Error The value of *x* is zero. 2

52449 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
52450 then *errno* shall be set to [ERANGE]. If the integer expression
52451 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the divide-by- 2
52452 zero floating-point exception shall be raised. 2

52453 Range Error The correct result would cause overflow. 2

52454 If the integer expression (*math_errhandling* & MATH_ERRNO) is non-zero,
52455 then *errno* shall be set to [ERANGE]. If the integer expression 2
52456 (*math_errhandling* & MATH_ERREXCEPT) is non-zero, then the overflow 2
52457 floating-point exception shall be raised. 2

52458	Range Error	The value of x is too large in magnitude, or the correct result would cause underflow.
52459		
52460		If the integer expression (<code>math_errhandling & MATH_ERRNO</code>) is non-zero, then <code>errno</code> shall be set to [ERANGE]. If the integer expression (<code>math_errhandling & MATH_ERREXCEPT</code>) is non-zero, then the underflow floating-point exception shall be raised.
52461		
52462		
52463		

52464 EXAMPLES

52465 None.

52466 APPLICATION USAGE

52467 On error, the expressions (`math_errhandling & MATH_ERRNO`) and (`math_errhandling & MATH_ERREXCEPT`) are independent of each other, but at least one of them must be non-zero.

52469 RATIONALE

52470 None.

52471 FUTURE DIRECTIONS

52472 None.

52473 SEE ALSO

52474 *feclearexcept()*, *fetestexcept()*, *isnan()*, *j0()*, the Base Definitions volume of IEEE Std 1003.1-2001,
52475 Section 4.18, Treatment of Error Conditions for Mathematical Functions, <**math.h**>

52476 CHANGE HISTORY

52477 First released in Issue 1. Derived from Issue 1 of the SVID.

52478 Issue 5

52479 The DESCRIPTION is updated to indicate how an application should check for an error. This
52480 text was previously published in the APPLICATION USAGE section.

52481 Issue 6

52482 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

52483 The RETURN VALUE and ERRORS sections are reworked for alignment of the error handling
52484 with the ISO/IEC 9899:1999 standard.

52485 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/148 is applied, updating the RETURN 2
52486 VALUE and ERRORS sections. The changes are made for consistency with the general rules 2
52487 stated in “Treatment of Error Conditions for Mathematical Functions” in the Base Definitions 2
52488 volume of IEEE Std 1003.1-2001. 2

Index

_CS_PATH	219
_CS_XBS5_ILP32_OFF32_CFLAGS	219
_CS_XBS5_ILP32_OFF32_LDFLAGS	219
_CS_XBS5_ILP32_OFF32_LIBS	219
_CS_XBS5_ILP32_OFFBIG_CFLAGS	219
_CS_XBS5_ILP32_OFFBIG_LDFLAGS	219
_CS_XBS5_ILP32_OFFBIG_LIBS	219
_CS_XBS5_LP64_OFF64_CFLAGS	219
_CS_XBS5_LP64_OFF64_LDFLAGS	219
_CS_XBS5_LP64_OFF64_LIBS	219
_CS_XBS5_LPBIG_OBBIG_CFLAGS	219
_CS_XBS5_LPBIG_OBBIG_LDFLAGS	219
_CS_XBS5_LPBIG_OBBIG_LIBS	219
_exit	87, 313, 1612
_Exit()	313, 87
FILE	131
_IOFBF	1303, 1345
_IOLBF	384, 1345
_IONBF	1303, 1345
LINE	131
_longjmp()	88
_LVL	17
_MAX	16
_MIN	16
_PC constants	
used in pathconf	407
_PC_ALLOC_SIZE_MIN	407
_PC_ASYNC_IO	407
_PC_CHOWN_RESTRICTED	407
_PC_FILESIZEBITS	407
_PC_LINK_MAX	407
_PC_MAX_CANON	407
_PC_MAX_INPUT	407
_PC_NAME_MAX	407
_PC_NO_TRUNC	407
_PC_PATH_MAX	407
_PC_PIPE_BUF	407
_PC_PRIO_IO	407
_PC_REC_INCR_XFER_SIZE	407
_PC_REC_MAX_XFER_SIZE	407
_PC_REC_MIN_XFER_SIZE	407
_PC_REC_XFER_ALIGN	407
_PC_SYMLINK_MAX	407
_PC_SYNC_IO	407
_PC_VDISABLE	407
_POSIX2 constants	
in sysconf	1492
_POSIX2_CHAR_TERM	1494
_POSIX2_C_BIND	1494
_POSIX2_C_DEV	1494
_POSIX2_FORT_DEV	1494
_POSIX2_FORT_RUN	1494
_POSIX2_LOCALEDEF	1494
_POSIX2_PBS	1494
_POSIX2_PBS_ACCOUNTING	1494
_POSIX2_PBS_LOCATE	1494
_POSIX2_PBS_MESSAGE	1494
_POSIX2_PBS_TRACK	1494
_POSIX2_SW_DEV	1494
_POSIX2_UPE	1494
_POSIX2_VERSION	1494
_POSIX	15, 17
_POSIX_ADVISORY_INFO	1492
_POSIX_ASYNCRONOUS_IO	1492
_POSIX_ASYNC_IO	407
_POSIX_BARRIERS	1492
_POSIX_CHOWN_RESTRICTED	195, 407, 409
_POSIX_CLOCK_SELECTION	1492
_POSIX_CPUTIME	1492
_POSIX_C_SOURCE	14
_POSIX_FSYNC	1493
_POSIX_JOB_CONTROL	1493
_POSIX_MAPPED_FILES	1493
_POSIX_MEMLOCK	1493
_POSIX_MEMLOCK_RANGE	1493
_POSIX_MEMORY_PROTECTION	1493
_POSIX_MESSAGE_PASSING	1493
_POSIX_MONOTONIC_CLOCK	1493
_POSIX_NO_TRUNC	407
_POSIX_OPEN_MAX	565
_POSIX_PRIORITIZED_IO	42, 1493
_POSIX_PRIORITY_SCHEDULING	42, 1493
_POSIX_PRIO_IO	407
_POSIX_READER_WRITER_LOCKS	1493
_POSIX_REALTIME_SIGNALS	1493
_POSIX_REGEX	1493
_POSIX_SAVED_IDS	1493
_POSIX_SEMAPHORES	1493
_POSIX_SHARED_MEMORY_OBJECTS	1493
_POSIX_SHELL	1493
_POSIX_SOURCE	14

_POSIX_SPAWN	1493
_POSIX_SPIN_LOCKS	1493
_POSIX_SPORADIC_SERVER	1493
_POSIX_SYNCHRONIZED_IO	1493
_POSIX_SYNC_IO	407
_POSIX_THREADS	246, 1493, 1552
_POSIX_THREAD_ATTR_STACKADDR	1493
_POSIX_THREAD_ATTR_STACKSIZE	1493
_POSIX_THREAD_CPUTIME	1493
_POSIX_THREAD_PRIORITY_SCHEDULING	1493
_POSIX_THREAD_PRIO_INHERIT	1493
_POSIX_THREAD_PRIO_PROTECT	1493
_POSIX_THREAD_PROCESS_SHARED	1107 1493
_POSIX_THREAD_SAFE_FUNCTIONS	246 1493, 1552
_POSIX_THREAD_SPORADIC_SERVER	1493
_POSIX_TIMEOUTS	1493
_POSIX_TIMERS	1493
_POSIX_TRACE	1493
_POSIX_TRACE_EVENT_FILTER	1493
_POSIX_TRACE_EVENT_NAME_MAX	952, 954
_POSIX_TRACE_INHERIT	1493
_POSIX_TRACE_LOG	1493
_POSIX_TRACE_SYS_MAX	949
_POSIX_TRACE_USER_EVENT_MAX	954
_POSIX_TYPED_MEMORY_OBJECTS	1493
_POSIX_V6_ILP32_OFF32	1493
_POSIX_V6_ILP32_OFFBIG	1494
_POSIX_V6_LP64_OFF64	1494
_POSIX_V6_LPBIG_OFFBIG	1494
_POSIX_VDISABLE	407
_POSIX_VERSION	1493, 1576
_PROCESS	17
_PTHREAD_THREADS_MAX	1081
_SC constants	
in sysconf	1492
_SC_2_CHAR_TERM	1494
_SC_2_C_BIND	1494
_SC_2_C_DEV	1494
_SC_2_FORT_DEV	1494
_SC_2_FORT_RUN	1494
_SC_2_LOCALEDEF	1494
_SC_2_PBS_ACCOUNTING	1494
_SC_2_PBS_LOCATE	1494
_SC_2_PBS_MESSAGE	1494
_SC_2_PBS_TRACK	1494
_SC_2_SW_DEV	1494
_SC_2_UPE	1494
_SC_2_VERSION	875, 1494
_SC_ADVISORY_INFO	1492
_SC_AIO_LISTIO_MAX	1492
_SC_AIO_MAX	1492
_SC_AIO_PRIO_DELTA_MAX	1492
_SC_ARG_MAX	1492
_SC_ASYNCHRONOUS_IO	1492
_SC_ATEXIT_MAX	1492
_SC_BARRIERS	1492
_SC_BC_BASE_MAX	1492
_SC_BC_DIM_MAX	1492
_SC_BC_SCALE_MAX	1492
_SC_BC_STRING_MAX	1492
_SC_CHILD_MAX	1492
_SC_CLK_TCK	1492, 1548
_SC_CLOCK_SELECTION	1492
_SC_COLL_WEIGHTS_MAX	1492
_SC_CPUTIME	1492
_SC_DELAYTIMER_MAX	1492
_SC_EXPR_NEST_MAX	1492
_SC_FSYNC	1493
_SC_GETGR_R_SIZE_MAX	516, 519, 1492
_SC_GETPW_R_SIZE_MAX	557, 560, 1492
_SC_IOV_MAX	1492
_SC_JOB_CONTROL	1493
_SC_LINE_MAX	1492
_SC_LOGIN_NAME_MAX	1492
_SC_MEMLOCK	1493
_SC_MEMLOCK_RANGE	1493
_SC_MEMORY_PROTECTION	1493
_SC_MESSAGE_PASSING	1493
_SC_MONOTONIC_CLOCK	1493
_SC_MQ_OPEN_MAX	1492
_SC_MQ_PRIO_MAX	1492
_SC_NGROUPS_MAX	1492
_SC_OPEN_MAX	1492
_SC_PAGESIZE	787, 881, 1494
_SC_PAGE_SIZE	787, 1494
_SC_PRIORITIZED_IO	1493
_SC_PRIORITY_SCHEDULING	1493
_SC_READER_WRITER_LOCKS	1493
_SC_REALTIME_SIGNALS	1493
_SC_REGEX	1493
_SC_RE_DUP_MAX	1494
_SC_RTSIG_MAX	1494
_SC_SAVED_IDS	1493
_SC_SEMAPHORES	1493
_SC_SEM_NSEMS_MAX	1494
_SC_SEM_VALUE_MAX	1494
_SC_SHARED_MEMORY_OBJECTS	1493
_SC_SHELL	1493
_SC_SIGQUEUE_MAX	1494

_SC_SPAWN	1493	_tolower()	90
_SC_SPIN_LOCKS	1493	_toupper()	91
_SC_SPORADIC_SERVER	1493	_XBS5_ILP32_OFF32	1494
_SC_STREAM_MAX	1494	_XBS5_ILP32_OFFBIG	1494
_SC_SYMLOOP_MAX	1494	_XBS5_LP64_OFF64	1494
_SC_SYNCHRONIZED_IO	1493	_XBS5_LPBIG_OFFBIG	1494
_SC_THREADS	1493	_XOPEN_CRYPT	1494
_SC_THREAD_ATTR_STACKADDR	1493	_XOPEN_ENH_I18N	1494
_SC_THREAD_ATTR_STACKSIZE	1493	_XOPEN_LEGACY	1494
_SC_THREAD_CPUTIME	1493	_XOPEN_REALTIME	345, 1494
_SC_THREAD_DESTRUCTOR_ITERATIONS	1494	_XOPEN_REALTIME_THREADS	1494
.....	1494	_XOPEN_SHM	1494
_SC_THREAD_KEYS_MAX	1494	_XOPEN_SOURCE	14
_SC_THREAD_PRIORITY_SCHEDULING	1493	_XOPEN_UNIX	1494
_SC_THREAD_PRIO_INHERIT	1493	_XOPEN_VERSION	1494
_SC_THREAD_PRIO_PROTECT	1493	a64l()	92
_SC_THREAD_PROCESS_SHARED	1493	ABDAY_1	846
_SC_THREAD_SAFE_FUNCTIONS	1493	abort()	94
_SC_THREAD_SPORADIC_SERVER	1493	abs()	96
_SC_THREAD_STACK_MIN	1494	accept()	97
_SC_THREAD_THREADS_MAX	1494	access()	99
_SC_TIMEOUTS	1493	acos()	102
_SC_TIMERS	1493	acosf()	102
_SC_TIMER_MAX	1494	acosh()	104
_SC_TRACE	1493	acoshf()	104
_SC_TRACE_EVENT_FILTER	1493	acosl()	104
_SC_TRACE_INHERIT	1493	ACTION	593
_SC_TRACE_LOG	1493	address information	435
_SC_TTY_NAME_MAX	1494	address string	435
_SC_TYPED_MEMORY_OBJECTS	1493	addrinfo structure	435
_SC_TZNAME_MAX	1494	ADV	3
_SC_V6_ILP32_OFF32	1493	ADVANCED REALTIME	201, 205, 812-813, 877
_SC_V6_ILP32_OFFBIG	1494	879, 881, 883, 885, 888, 896, 899
_SC_V6_LP64_OFF64	1494	901-903, 905, 907, 909, 911, 913
_SC_V6_LPBIG_OFFBIG	1494	915, 917-924, 977, 979, 1058, 1103, 1277
_SC_VERSION	1493	ADVANCED REALTIME THREADS	1026, 1028
_SC_XBS5_ILP32_OFF32	1494	1030, 1032, 1034-1035, 1075, 1159
_SC_XBS5_ILP32_OFFBIG	1494	1161, 1163
_SC_XBS5_LP64_OFF64	1494	AF	18
_SC_XBS5_LPBIG_OFFBIG	1494	AIO	3
_SC_XOPEN_CRYPT	1494	AIO_	16
_SC_XOPEN_ENH_I18N	1494	aio_	16
_SC_XOPEN_LEGACY	1494	AIO_ALLDONE	107
_SC_XOPEN_REALTIME	1494	aio_cancel()	107
_SC_XOPEN_REALTIME_THREADS	1494	AIO_CANCELED	107
_SC_XOPEN_SHM	1494	aio_error()	109
_SC_XOPEN_UNIX	1494	aio_fsync()	110
_SC_XOPEN_VERSION	1494	AIO_LISTIO_MAX	696, 1492
_setjmp	88	AIO_MAX	696, 1492
_t	17	AIO_NOTCANCELED	107
_TIME	17		

AIO_PRIO_DELTA_MAX	42, 1492
aio_read()	112
aio_return()	115
aio_suspend()	116
aio_write()	118
AI_ALL	436
AI_CANONNAME	436
AI_INET6	436
AI_NUMERICHOST	436
AI_NUMERICSERV	436
AI_PASSIVE	436
AI_V4MAPPED	436
alarm()	121
anycast	68
ANYMARK	629
appropriate privileges	100, 408
argc	308
ARG_MAX	22, 302, 305, 310, 1492
asctime()	123
asctime_r()	123
asin()	126
asinf	126
asinh()	128
asinhf()	128
asinhl()	128
asinl()	126, 130
assert()	131
async-signal-safe	992
atan()	132
atan2()	134
atan2f()	134
atan2l()	134
atanf()	132, 137
atanh()	138
atanhf()	138
atanhl()	138
atnl()	132, 140
atexit()	141
ATEXIT_MAX	141, 1492
atof()	143
atoi()	144
atol()	146
atoll()	146
attributes, clock-resolution	77, 927
attributes, creation-time	77, 927
attributes, generation-version	76, 927
attributes, inheritance	77, 929
attributes, log-full-policy	75, 77, 929, 932
attributes, log-max-size	77, 930, 932
attributes, max-data-size	77, 932-933
attributes, stream-full-policy	73-74, 77, 930
attributes, stream-min-size	77, 933
attributes, trace-name	77, 927
attributes, truncation-status	952
background	1323
background process	1523
BAR	3
basename()	147
baud rate functions	185
bcmp()	149
bcopy()	150
BC_constants	
in sysconf	1492
BC_BASE_MAX	1492
BC_DIM_MAX	1492
BC_SCALE_MAX	1492
BC_STRING_MAX	1492
BE	4
bind()	151
bi	16
BOOT_TIME	289-290
broadcasting a condition	1044
BSD	121, 196, 316, 342, 409
.....	548, 678, 767, 835, 1194, 1234
.....	1242, 1323, 1367, 1391, 1515
.....	1538, 1576, 1612
bsd_signal()	153
bsearch()	155
btowc()	158
buffer cache	461
BUFSIZ	1303
BUS	18
byte-oriented stream	36
byte-stream mode	1191
bzero()	159
cabs()	160
cabsf()	160
cabsl()	160
cacos()	161
cacosf()	161
cacosh()	162
cacoshf()	162
cacoshl()	162
cacosl()	161, 163
calloc()	164
can	1
cancel-safe	1149
cancelability state	1082, 1149
cancelability states	54
cancelability type	1082, 1149
canceling execution of a thread	1036
canonical name	436

carg()	166	clearerr()	199
cargf()	166	clock tick	121, 1495, 1548
cargl()	166	clock ticks/second	1492
casin()	167	clock()	200
casinf()	167	clock-resolution attribute	77, 927
casinh()	168	CLOCKS_PER_SEC	200
casinhf()	168	CLOCK_	17
casinhl()	168	clock	17
casinl()	167, 169	clock_getcpu_clockid()	201
catan()	170	clock_getres()	202
catanf()	170	clock_gettime()	202
catanh()	171	CLOCK_MONOTONIC	49, 206
catanhf()	171	clock_nanosleep()	205
catanhl()	171	CLOCK_PROCESS_CPUTIME_ID	49
catanl()	170, 172	CLOCK_REALTIME	49, 202, 206
catclose()	173	835, 1103, 1277, 1540
catgets()	174	clock_settime()	202, 208
catopen()	176	CLOCK_THREAD_CPUTIME_ID	50
CBAUD	19	clog()	209
cbrt()	178	clogf()	209
cbrtf()	178	clogl()	209
cbrtl()	178	close a file	212
ccos()	179	close()	210
ccosf()	179	closedir()	213
ccosh()	180	closelog()	215
ccoshf()	180	cmsg	16
ccoshl()	180	CMSG_	18
ccosl()	179, 181	COLL_WEIGHTS_MAX	1492
CD	4	command interpreter	
ceil()	182	portable	1612
ceilf()	182	compare thread IDs	1070
ceil(l)	182	compilation environment	13
cexp()	184	condition variable initialization attributes	1056
cexpf()	184	conforming application	1408
cexpl()	184	conforming application, strictly	121, 308
cfgetispeed()	185	confstr()	219
cfgetospeed()	187	conj()	222
cfsetispeed()	188	conjf()	222
cfsetospeed()	189	conjl()	222
change current working directory	191	connect()	223
change file modes	194	control data	38
change owner and group of file	196	control-normal	1191
CHAR_MAX	707, 709	conversion descriptor	302, 307, 599-602
chdir()	190	conversion specification	413, 444
CHILD_MAX	403, 1492	477, 486, 1443, 1447, 1460
chmod()	192	modified	1449
chown()	195	conversion specifier	
cimag()	198	modified	1461
cimagf()	198	copysign()	226
cimagl()	198	copysignf()	226
CLD_	18	copysignl()	226

core	1613	daylight	250, 1568
core file.....	315	DBL_MANT_DIG	182, 385
cos().....	227	DBL_MAX_EXP	182, 385
cosf()	227	DBM	251-252
cosh()	229	dbm_.....	16
coshf().....	229	DBM.....	18
coshl()	229	dbm_clearerr()	251
cosl()	227, 231	dbm_close()	251
covert channel.....	678	dbm_delete()	251
cpow()	232	dbm_error()	251
cpowf()	232	dbm_fetch()	251
cpowl()	232	dbm_firstkey()	251
cproj()	233	DBM_INSERT	251
cprojf()	233	dbm_nextkey()	251
cprojl()	233	dbm_open()	251
CPT	4	DBM_REPLACE	251
creal()	234	dbm_store()	251
crealf()	234	DEAD_PROCESS	289-290
creal(l).....	234	DEFECCHO	19
creat()	235	deferred cancelability	1082
create a per-process timer.....	1541	delay process execution	1407
create an interprocess channel.....	869	DELAYTIMER_MAX	1492, 1545
create session and set process group ID.....	1335	dependency ordering	264
creation-time attribute	77, 927	descriptive name	435
CRYPT.....	237, 275, 1316	destroying a mutex	1092
crypt()	237	destroying condition variables	1048
CS	4	destructor functions	1086
csin()	239	detaching a thread	1068
csinf()	239	difftime()	255
csinh()	240	DIR	81, 213, 857, 1197, 1199, 1237, 1263, 1531
csinhf().....	240	directive	413, 444, 477, 486, 1460
csinhl()	240	directory operations	858
csinl()	239, 241	dirent structure	858
csqrt()	242	dirname()	256
csqrftf()	242	div()	258
csqrftl()	242	dlclose()	259
ctan()	243	dlerror()	261
ctanf()	243	dlopen()	263
ctanh()	244	dlsym()	266
ctanhf()	244	dot	858, 1234
ctanhl()	244	dot-dot	858, 1234
ctanl()	243, 245	drand48()	268
ctermid()	246	dup()	270
ctime()	248	dup2()	270
ctime_r().....	248	dynamic package initialization	1124
CX.....	4	d_	16
c_	17	E2BIG	22
data key creation	1086	EACCES	22
data messages.....	38	EADDRINUSE	22
data type	81	EADDRNOTAVAIL	22
DATEMSK	504	EAFNOSUPPORT	22

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	19	ENOLCK	25
ECHOKE	19	ENOLINK	25
ECHOPRT	19	ENOMEM	25
ECONNABORTED	23	ENOMSG	25
ECONNREFUSED	23	ENOPROTOOPT	25
ECONNRESET	23	ENOSPC	26
ecvt()	273	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	196, 309, 522	ENOTTY	26
effective user ID	100, 309, 678	ENTRY	593
EHOSTUNREACH	23	environ	292, 309
EIDRM	24	envp	309
Eighth Edition UNIX	1681	ENXIO	26
EILSEQ	24, 37	EOPNOTSUPP	26
EINPROGRESS	24, 42	EOVERFLOW	26
EINTR	24, 57, 946	EPERM	26
EINVAL	24, 933, 946	EPIPE	27
EIO	24	EPROTO	27
EISCONN	24	EPROTONOSUPPORT	27
EISDIR	24	EPROTOTYPE	27
ELOOP	24	erand48()	268, 293
ELSIZE	736	ERANGE	27
EMFILE	24	erf()	294
EMLINK	24	erfc()	296
EMPTY	290	erfcf()	296
EMSGSIZE	24	erfc1()	296
EMULTIHOP	25	erff()	294, 298
ENAMETOOLONG	25	erfl()	294, 298
encrypt()	275	EROFS	27
endgrent()	277	errno	299
endhostent()	279	error descriptions	492
endnetent()	281	error numbers	21
endprotoent()	283	additional	28
endpwent()	285	ESPIPE	27
endservent()	287	ESRCH	27
endutxent()	289	EST5EDT	1568
ENETDOWN	25	ESTALE	27
ENETRESET	25	ETIME	27

ETIMEDOUT	27
ETXTBSY	27
EWOULDBLOCK	27
examine and change blocked signals	1157
examine and change signal action	1366
EXDEV	28
exec	301
of shell scripts	308
exec family.	100, 212, 341, 382, 405, 992, 1324, 1612
execel()	301
execle()	301
execlp()	301
execute a file	308
execution time monitoring	49
execv()	301
execve()	301
execvp()	301
exit()	313
EXIT_FAILURE	313
EXIT_SUCCESS	313, 316
exp()	318
exp2()	320
exp2f()	320
exp2l()	320
expf()	318
expl()	318
expm1()	322
expm1f()	322
expm1l()	322
EXPR_NEST_MAX	1492
EXTA	19
EXTB	19
extension	
CX	4
OH	7
XSI	11
extensions to setlocale	1318
fabs()	324
fabsf()	324
fabsl()	324
fattach()	326
fchdir()	329
fchmod()	330
fchown()	332
fclose()	334
fcntl()	336
fcvt()	273, 344
FD	4
fdatsync()	345
fdetach()	346
fdim()	348
fdimf()	348
fdiml()	348
fdopen()	350
fds_	16
FD_	16, 18
fd_	16
FD_CLOEXEC	35, 176, 302, 336602, 850, 858, 868, 889, 896, 1347
FD_CLR()	987, 86
FD_ISSET	86, 987
FD_SET	86, 987
FD_ZERO	86, 987
feature test macro	13, 497 _POSIX_C_SOURCE
_XOPEN_SOURCE	14
feclearexcept()	352
fegetenv()	353
fegetexceptflag()	354
fegetround()	355
feholdexcept()	357
feof()	358
feraiseexcept()	359
ferror()	360
fesetenv()	353, 361
fesetexceptflag()	354, 362
fesetround()	355, 363
fetestexcept()	364
feupdateenv()	366
fflush()	368
ffs()	371
fgetc()	372
fgetpos()	374
fgets()	376
fgetwc()	378
fgetws()	380
FIFO	769-770, 854, 1680
FILE	81, 199, 334, 350, 358360, 368, 372, 374, 376, 378380, 382-383, 398, 413, 425427, 429, 431-432, 439, 444451, 454, 463, 475-477, 484486, 495-496, 582, 864, 8741166-1167, 1179, 1236, 13031345, 1428, 1550, 1578-15791596, 1598, 1600, 1602, 1604, 1606
file	
locking	341
file accessibility	100
file control	341
FILE object	34
file permission bits	100

file permissions	100, 409, 1425	fputws().....	431
file position indicator.....	34	FQDN.....	537
fileno()	382	FR	4
FILESIZEBITS.....	407	fread().....	432
FIND.....	593	free().....	434
find string token.....	1473	freeaddrinfo().....	435
flockfile().....	383	freopen()	439
floor()	385	frexp().....	442
floorf().....	385	frexpf().....	442
floorl().....	385	frexpl().....	442
FLT_RADIX.....	726	FSC	5
FLT_ROUNDS.....	387	fscanf().....	444
FLUSH.....	18	fseek()	451
FLUSHO	19	fseeko().....	451
FLUSHR.....	623	fsetpos()	454
FLUSHRW	623	fstat().....	456
FLUSHW	623	fstatvfs()	458
fma()	387	fsync().....	461
fmaf()	387	ftell()	463
fmal()	387	ftello()	463
fmax()	389	ftime()	465
fmaxf()	389	ftok()	467
fmaxl()	389	ftruncate()	469
fmin()	390	ftrylockfile()	383, 471
fminf()	390	FTW	18, 841-842
fminl()	390	ftw()	472
FMNAMESZ.....	622	FTW_CHDIR	841
fmod().....	391	FTW_D	472, 841
fmodf()	391	FTW_DEPTH	841
fmodl().....	391	FTW_DNR	472, 841-842
fmtmsg()	393	FTW_DP	841
fnmatch()	396	FTW_F	472, 841
FNM_	18	FTW_MOUNT	841
FNM_NOESCAPE.....	396	FTW_NS	472, 841-842
FNM_NOMATCH.....	396	FTW_PHYS	841
FNM_PATHNAME.....	396	FTW_SL	472, 841
FNM_PERIOD.....	396	FTW_SLN	841
fopen()	398	fully-qualified domain name	537
FOPEN_MAX.....	350, 399, 874, 1550	functions.....	13
foreground	1323	implementation.....	13
fork handler	993	use.....	13
fork()	402	funlockfile().....	383, 475
forkall	405	fwide()	476
format of entries.....	11	fwprintf()	477
fpathconf()	407	fwrite()	484
fpclassify()	412	fwscanf()	486
FPE_	18	f_	16
fprintf()	413	F_	18
fputc()	425	F_DUPFD	270, 336, 338-339
fputs()	427	F_GETFD	336, 338, 341
fputwc()	429	F_GETFL	336, 338, 341

F_GETLK	337-339	getlogin().....	530
F_GETOWN	336, 338	getlogin_r().....	530
F_LOCK	715	getmsg()	533
F_RDLCK	339	getnameinfo()	537
F_SETFD	336, 338, 341	GETNCNT	1284-1285
F_SETFL	336, 338, 341	getnetbyaddr().....	281, 540
F_SETLK	337-339	getnetbyname()	281, 540
F_SETLKW	55, 337-339	getnetent()	281, 540
F_SETOWN	336, 339	getopt().....	541
F_TEST	715	getpeername().....	546
F_TLOCK	715	getpgid()	547
F_ULOCK	715	getpgrp()	548
F_UNLCK	337-338	GETPID	1284-1285
F_WRLCK	339	getpid().....	549
gai_strerror()	492	getpmsg()	533, 550
gcvt()	273, 493	getppid()	551
generation-version attribute	76, 927	getpriority()	552
get configurable pathname variables	409	getprotent()	555
get configurable system variables.....	1495	getprotobyname().....	283, 555
get file status.....	1425	getprotobynumber()	283, 555
get process times	1548	getprotoent()	283
get supplementary group IDs.....	521	getpwent().....	285, 556
get system time	1538	getpwnam()	557
get thread ID.....	1147	getpwnam_r()	557
get user name	531	getpwuid()	560
getaddrinfo()	435, 494	getpwuid_r()	560
GETALL.....	1284	getrlimit().....	563
getc()	495	getrusage()	566
getchar()	498	gets()	568
getchar_unlocked()	496, 499	getservbyname()	287, 569
getcontext()	500	getservbyport()	287, 569
getcwd()	502	getserver()	287, 569
getc_unlocked()	496	getsid()	570
getdate()	504	getsockname()	571
getdate_err().....	504	getsockopt()	572
getegid()	509	getsubopt()	575
getenv()	309, 510	gettimeofday()	579
geteuid()	513	getuid()	580
getgid()	514	getutxent()	289, 581
getgrent()	277, 515	getutxid()	289, 581
getgrgid()	516	getutxline()	289, 581
getgrgid_r()	516	GETVAL	1284-1285
getgrnam()	519	getwc()	582
getgrnam_r()	519	getwchar()	583
getgroups()	521	getwd()	503, 584
gethostbyaddr()	523	GETZCNT	1284-1285
gethostbyname()	523	glob()	585
gethostent()	279, 525	globfree()	585
gethostid()	526	GLOB_	18
gethostname()	527	GLOB_constants	
getitimer()	528	error returns of glob.....	587

used in glob	585	ILL_.....	18
GLOB_ABORTED	587	ilogb()	608
GLOB_APPEND	585-586	ilogbf()	608
GLOB_DOOFFS	585-586	ilogbl()	608
GLOB_ERR	585, 587	imaxabs()	610
GLOB_MARK	586	imaxdiv()	611
GLOB_NOCHECK	586-587	implementation-defined	1
GLOB_NOESCAPE	586	IMPLINK_.....	18
GLOB_NOMATCH	587	in6_.....	16
GLOB_NOSORT	586	IN6_.....	18
GLOB_NOSPACE	587	INADDR_.....	18
gl	16	index()	612
GMT0	1568	inet_.....	16
gmtime()	589	inet_addr()	613
gmtime_r().....	589	inet_ntoa()	613
grantpt()	591	inet_ntop()	615
granularity of clock	465	inet_pton()	615
HALT	394	Inf	126
hcreate()	593	INF	416, 480
hdestroy()	593	INFINITY	416, 480
high resolution sleep	835	INFO	394
host name	435	infu_.....	16
htonl()	596	inheritance attribute	77, 929
hton(s)	596	init	316, 678
HUGE_VAL	138, 182, 229, 318	initialize a named semaphore	1273
.....	320, 322, 348, 385, 597, 686	initialize an unnamed semaphore	1270
.....	690, 718, 722, 724, 837, 839	initializing a mutex	1092
.....	982, 1239, 1244, 1248, 1404	initializing condition variables	1048
.....	1468, 1504, 1535, 1639	initstate()	617
HUGE_VALF	385, 1239, 1468	INIT_PROCESS	289-290
HUGE_VALL	385, 1239, 1468	input and output rationale	1193
hypot()	597	insque()	619
hypotf()	597	international environment	1318
hypotl()	597	Internet Protocols	67
h_.....	16	INT_MAX	608
h_errno	592	INT_MIN	96
iconv()	599	in_.....	16
iconv_close()	601	IN_.....	18
iconv_open()	602	ioctl()	622
ic_.....	16	iov_.....	17
IEEE Std 754-1985	3	IOV_.....	18
IEEE Std 854-1987	3	IOV_MAX	1204, 1492, 1684
ifc_.....	16	IP6	5
ifra_.....	16	IPC	39 , 818, 820, 823, 825, 1289, 1293, 1357, 1359
ifru_.....	16	ipc_.....	16
if_.....	16	IPC_.....	18
IF_	18	IPC_constants	
if_freenameindex()	604	used in semctl	1284
if_indextoname()	605	used in shmctl	1355
if_nameindex()	606	IPC_CREAT	819, 1287, 1358
if_nametoindex()	607	IPC_EXCL	819, 1287

IPC_NOWAIT	821-822, 824-825, 1290	1391, 1538
IPC_PRIVATE	819, 1287, 1358	654
IPC_RMID	817, 1285, 1355	655
IPC_SET	817, 1285, 1355	656
IPC_STAT	817, 1284, 1355	657
IPPORT_.....	18	658
IPPROTO_.....	18	659
IPv4	68	660
IPv4-compatible address.....	69	661
IPv4-mapped address.....	69	662
IPv6	68	663
compatibility with IPv4.....	69	665
interface identification.....	69	666
options	70	667
IPv6 address		
anycast.....	68	668
loopback.....	69	669
multicast.....	68	670
unicast.....	68	671
unspecified.....	69	672
IPV6	18	673
IPV6_JOIN_GROUP	70	528
IPV6_LEAVE_GROUP	70	528
IPV6_MULTICAST_HOPS	70	528
IPV6_MULTICAST_IF	70	528
IPV6_MULTICAST_LOOP	70	528
IPV6_UNICAST_HOPS	70	528
IPV6_V6ONLY	70	528
ip_	16	16-17
IP_	18	18
isalnum()	634	628-629
isalpha()	635	629
isascii()	636	629
isastream()	637	629
isatty()	638	629
isblank()	639	629
iscntrl()	640	629
isdigit()	641	629
isfinite()	642	629
isgraph()	643	629
isgreater()	644	629
isgreaterequal()	645	629
isinf()	646	629
isless()	647	629
islessequal()	648	629
islessgreater()	649	629
islower()	650	22, 628
isnan()	652	627-628
isnormal()	653	210, 629
ISO C standard.....	3, 121, 308, 341, 497	623-624
738, 1187, 1234, 1318, 1367		624-625, 1191

I_STR	626	list directed I/O	697
I_SWROPT.....	627, 1678	listen().....	699
I_UNLINK.....	630	llabs().....	682, 701
j0().....	674	lldiv().....	688, 702
j1().....	674	LLONG_MAX.....	1476, 1645
jn()	674	LLONG_MIN.....	1476
job control	316, 548, 678, 1323, 1335, 1495, 1612	llrint().....	703
jrand48().....	268, 676	llrintf().....	703
JST-9	1568	llrintl().....	703
kill().....	677	llround().....	705
killpg().....	680	llroundf().....	705
l64a()	92, 681	llroundl().....	705
labs()	682	load ordering.....	264
LANG.....	176	LOBLK	19
last close	1351	localeconv().....	707
LASTMARK.....	629	localtime().....	712
lchown()	683	localtime_r().....	712
lccong48()	268, 685	lockf().....	715
LC_ALL.....	302, 709, 846, 1317, 1319	locking.....	341
LC_COLLATE.....	585-586, 1317, 13191435, 1483, 1623, 1657	advisory.....	341
LC_CTYPE.....	158, 663, 747, 749, 751753-754, 756, 758, 1317, 13191555-1559, 1618, 1634, 16491659-1660, 1662-1663	mandatory.....	341
LC_MESSAGES.....	176, 1317-1319, 1441	locking and unlocking a mutex.....	1100
LC_MONETARY.....	709, 1317, 1319, 1444	log().....	718
LC_NUMERIC.....	273, 414, 444, 477, 486709, 1317, 1319, 1444, 1468, 1639	log-full-policy attribute.....	75, 77, 929, 932, 950
LC_TIME.....	505, 846, 1317, 1319	log-max-size attribute.....	77, 930, 932
ldexp()	686	log10().....	720
ldexpf().....	686	log10f().....	720
ldexpl().....	686	log10l().....	720
ldiv()	688	log1p().....	722
legacy.....	1	log1pf().....	722
lfind().....	736, 689	log1pl().....	722
lgamma().....	690	log2().....	724
lgammaf()	690	log2f().....	724
lgammal()	690	log2l().....	724
LINE_MAX.....	1492	logb().....	726
link to a file.....	694	logbf().....	726
link().....	692	logbl().....	726
LINK_MAX.....	24, 407, 692, 1233	logf().....	718, 728
LIO_.....	16	login shell.....	308
lio_	16	LOGIN_NAME_MAX.....	530, 1492
lio_listio().....	695	LOGIN_PROCESS.....	289-290
LIO_NOP.....	695	logl().....	718, 728
LIO_NOWAIT.....	695	LOG_.....	18
LIO_READ.....	695	LOG_constants in syslog.....	215
LIO_WAIT.....	695	LOG_ALERT	215
LIO_WRITE	695	LOG_CONS	216
		LOG_CRIT	215
		LOG_DEBUG	215
		LOG_EMERG	215
		LOG_ERR	215
		LOG_INFO.....	215

LOG_LOCAL.....	215	MCL_FUTURE.....	783
LOG_NDELAY.....	216	memccpy().....	760
LOG_NOTICE.....	215	memchr()	761
LOG_NOWAIT.....	216	memcmp()	762
LOG_ODELAY.....	216	memcpy().....	763
LOG_PID.....	216	MEMLOCK_FUTURE.....	790
LOG_USER.....	215-216	memmove().....	764
LOG_WARNING.....	215	memory management.....	43
longjmp()	729	memory protection option	790
LONG_MAX.....	1476, 1645	memset()	765
LONG_MIN.....	1476, 1645	message catalog descriptor.....	302, 307, 313
lrand48	268	message parts	39
lrand48().....	731	message priority	38
lrint().....	732	high-priority	38
lrintf()	732	normal.....	38
lrintl().....	732	priority	38
lround()	734	message-discard mode.....	1191
lroundf().....	734	message-nondiscard mode.....	1191
lroundl().....	734	MET-1MEST	1568
lsearch().....	736	MF.....	5
lseek()	738	MINSIGSTKSZ.....	1370
lstat().....	740	mkdir()	766
l_.....	16	mkfifo()	769
L_ctermid	246	mknod()	772
l_sysid	341	mkstemp()	775
makecontext().....	742	mktemp()	777
malloc()	745	mktimedate()	779
manipulate signal sets.....	1373	ML.....	6
mappings.....	790	mlock()	781
MAP_.....	16, 18	mlockall().....	783
MAP_FAILED	791	MLR.....	6
MAP_FIXED.....	786, 789	mmap().....	785
MAP_PRIVATE.....	402, 786, 790, 795, 827	MM.....	18
MAP_SHARED.....	405, 786-787	MM_APPL.....	393
max-data-size attribute	77, 932-933	MM_CONSOLE.....	393
MAX_CANON.....	407	MM_ERROR	394-395
MAX_INPUT.....	407	mm_FIRM	393
may	2	MM_HALT	394
mblen()	747	MM_HARD.....	393
mbrlen()	749	MM_INFO	394
mbrtowc().....	751	MM_NOCON	394
mbsinit().....	753	MM_NOMSG	394
mbsrtowcs()	754	MM_NOSEV	394
mbstowcs().....	756	MM_NOTOK	394
mbtowc().....	758	MM_NRECOV	393
MB_CUR_MAX.....	747, 749, 751, 758, 1618, 1660	MM_NULLMC	393
MC1	5	MM_OK	394
MC2	5	MM_OPSYS	393
MC3	5	MM_PRINT	393, 395
MCL.....	16	MM_RECOVER	393
MCL_CURRENT	783	MM_SOFT	393

MM_UTIL.....	393
MM_WARNING.....	394
modf().....	793
modff().....	793
modfl().....	793
MON.....	6
MORECTL.....	534
MOREDATA.....	534
MPR.....	6
mprotect().....	795
MQ_.....	16
mq_.....	16
mq_close()	797
mq_getattr()	798
mq_notify().....	800
mq_open()	802
MQ_OPEN_MAX.....	1492
MQ_PRIO_MAX.....	808-809, 1492
mq_receive()	805
mq_send().....	808
mq_setattr().....	810
mq_timedreceive()	805, 812
mq_timedsend().....	808, 813
mq_unlink().....	814
mrand48().....	268, 816
MSG.....	6, 18
msgctl().....	817
msgget().....	819
msgrecv().....	821
msgsnd().....	824
MSGVERB.....	394-395
msg_.....	16
MSG_.....	18
MSG_ANY.....	533
MSG_BAND.....	533, 1172
MSG_EOR.....	1412, 1414
MSG_HIPRI.....	533, 1172
MSG_NOERROR.....	821-822
msg_perm.....	40
msqid.....	40
MST7MDT.....	1568
msync().....	827
MS_.....	16, 18
MS_ASYNC.....	787, 827
MS_INVALIDATE.....	827-828
MS_SYNC.....	787, 827
multicast.....	68
munlock().....	781, 830
munlockall().....	783, 831
munmap().....	832
mutex attributes.....	1107
mutex initialization attributes	1106
mutex performance.....	1107
MUXID_ALL.....	630-631
MUXID_R.....	18
MX.....	6
M_.....	18
name information.....	537
name space.....	14
NAME_MAX.....	25, 99, 176, 190, 192195, 306, 326, 346, 399, 407440, 458, 467, 683, 692, 740766, 769, 773, 803, 814, 842852, 857, 980, 1197, 1201, 12081233, 1241, 1273, 1281, 13481351, 1423, 1489, 1561, 15811589, 1591
NaN.....	126, 416, 480
NAN.....	416, 480
nan().....	834
nanf().....	834
nanl().....	834
nanosleep().....	835
NDEBUG.....	21, 131
nearbyint().....	837
nearbyintl().....	837
nearbyintf().....	837
nearbyintl().....	837
network interfaces.....	60
NEW_TIME.....	289-290
nextafter().....	839
nextafterf().....	839
nextafterl().....	839
nexttoward().....	839
nexttowardf().....	839
nexttowardl().....	839
nftw().....	841
NGROUPS_MAX.....	522, 1492
nice().....	844
NLSPATH.....	176
NL_.....	18
NL_ARGMAX.....	413, 444, 477, 486
NL_CAT_LOCALE.....	176
nl_langinfo().....	846
nohup utility.....	309
non-local jumps.....	1391
non-volatile storage.....	461
nrand48().....	268, 848
ntohl().....	596, 849
ntohs().....	596, 849
NULL.....	220, 246, 253, 261, 266, 790, 1199
NUM_EMPL.....	594
NZERO.....	552, 844

n.....	16
OB.....	6
obsolescent.....	185
OF.....	7
OH.....	7
OLD_TIME.....	289-290
open a file	854
open a named semaphore.....	1273
open a shared memory object.....	1349
open()	850
opendir()	857
openlog()	215, 860
OPEN_MAX.....	176, 270, 285, 339, 399439, 472, 516, 519, 530, 560, 602802, 842, 852, 857, 868, 896, 8991492, 1550
optarg.....	541, 861
opterr	541, 861
optind.....	541, 861
option	
ADV.....	3
AIO	3
BAR.....	3
BE.....	4
CD.....	4
CPT.....	4
CS.....	4
FD.....	4
FR.....	4
FSC.....	5
IP6.....	5
MC1.....	5
MC2.....	5
MC3.....	5
MF.....	5
ML.....	6
MLR.....	6
MON	6
MPR.....	6
MSG.....	6
MX.....	6
PIO.....	7
PS	7
RS.....	7
RTS.....	7
SD.....	7
SEM.....	8
SHM.....	8
SIO	8
SPI.....	8
SPN.....	8
SS.....	8
TCT.....	8
TEF.....	8
THR.....	9
TMO	9
TMR.....	9
TPI.....	9
TPP.....	9
TPS.....	9
TRC.....	9
TRI.....	10
TRL.....	10
TSA.....	10
TSF.....	10
TSH.....	10
TSP.....	10
TSS.....	10
TYM.....	10
UP	11
XSR	11
optopt	541, 544, 861
optstring	544
orphaned process group	316
O.....	18
O_constants	
used in open().....	850
used in posix_openpt().....	886
O_ACCMODE.....	336
O_APPEND.....	41, 118, 251, 351, 850, 1676
O_CREAT.....	235, 802-803, 814, 850-8521266, 1272, 1347-1349
O_DSYNC	110, 850-851, 1191, 1677
O_EXCL	803, 850-851, 1272, 1347-1348
O_NDELAY	1681
O_NOCTTY	851, 855, 886
O_NONBLOCK.....	24, 210, 334, 368, 372378, 425, 429, 452, 454, 534626, 628, 803, 805, 808, 810851-853, 868, 871, 1173, 11901677, 1680
O_RDONLY.....	256, 802, 850-852, 855, 1347, 1349
O_RDWR....	329, 715, 802, 850-854, 886, 1347, 1349
O_RSYNC	851, 1191
O_SYNC	110, 851, 1191, 1677
O_TRUNC	235, 851, 853, 855, 1348-1349
O_WRONLY.....	235, 329, 715, 802, 850-853, 855
PAGESIZE	43, 781, 828, 832, 999, 1494
PAGE_SIZE.....	1494
PATH.....	220
PATH environment variable	310
pathconf()	407, 862

PATH_MAX.....	25, 99, 176, 190, 192	195, 256, 306, 326, 346, 399	407, 440, 458, 467, 503, 584	683, 692, 740, 766, 769, 773	803, 814, 842, 852, 857, 980	1201, 1208, 1233, 1241, 1273	1281, 1348, 1351, 1423, 1489	1496, 1561, 1581, 1589, 1591	881
pause().....	863	POSIX_MADV_DONTNEED	881						
pclose().....	864	POSIX_MADV_NORMAL	881						
pd_.....	16	POSIX_MADV_RANDOM	881						
PENDIN.....	19	POSIX_MADV_SEQUENTIAL	881						
perror()	866	POSIX_MADV_WILLNEED	881						
persistent connection (L_PLINK).....	631	posix_memalign()	885						
PF_.....	18	posix_mem_offset()	883						
physical write	461	posix_openpt()	886						
ph_.....	16	POSIX_REC_INCR_XFER_SIZE	407						
PIO	7	POSIX_REC_MAX_XFER_SIZE	407						
pipe	404, 855, 1680	POSIX_REC_MIN_XFER_SIZE	407						
pipe()	868	POSIX_REC_XFER_ALIGN	407						
PIPE_BUF	407, 1677, 1680	posix_spawn()	888						
PIPE_MAX.....	1682	posix_spawnattr_destroy()	903						
plain characters.....	1443	posix_spawnattr_getflags()	905						
POLL	18	posix_spawnattr_getpgroup()	907						
poll()	870	posix_spawnattr_getschedparam()	909						
POLLERR	870	posix_spawnattr_getschedpolicy()	911						
POLLHUP	870	posix_spawnattr_getsigdefault()	913						
POLLIN	870	posix_spawnattr_getsigmask()	915						
POLLNVAL	871	posix_spawnattr_init()	903, 917						
POLLOUT	870	posix_spawnattr_setflags()	905, 918						
POLLPRI.....	870	posix_spawnattr_setpgroup()	907, 919						
POLLRDBAND	870	posix_spawnattr_setschedparam()	909, 920						
POLLRDNORM	870	posix_spawnattr_setschedpolicy()	911, 921						
POLLWRBAND	870	posix_spawnattr_setsigdefault()	913, 922						
POLLWRNORM	870	posix_spawnattr_setsigmask()	915, 923						
POLL_	18	posix_spawnp()	888, 924						
popen()	874	posix_spawn_file_actions_addclose()	896						
portability.....	3	posix_spawn_file_actions_adddup2()	899						
POSIX.....	273	posix_spawn_file_actions_addopen()	896, 901						
POSIX.1 symbols.....	13	posix_spawn_file_actions_destroy()	902						
POSIX_.....	15, 17	posix_spawn_file_actions_init()	902						
posix_.....	15, 17	POSIX_SPAWN_RESETIDS	889, 905						
POSIX_ALLOC_SIZE_MIN	407	POSIX_SPAWN_SETPGROUP	889, 905, 907						
posix_fadvise()	877	POSIX_SPAWN_SETSCHEDPARAM	905, 909						
POSIX_FADV_DONTNEED	877	POSIX_SPAWN_SETSCHEDULER	889, 905						
POSIX_FADV_NOREUSE	877	909, 911						
POSIX_FADV_NORMAL	877	POSIX_SPAWN_SETSIGDEF	890, 905, 913						
POSIX_FADV_RANDOM	877	POSIX_SPAWN_SETSIGMASK	905, 915						
POSIX_FADV_SEQUENTIAL	877	POSIX_TRACE_ADD_EVENTSET	963						
POSIX_FADV_WILLNEED	877	POSIX_TRACE_ALL_EVENTS	957						
posix_fallocate()	879	POSIX_TRACE_APPEND	930, 950						
		posix_trace_attr_destroy()	925						
		posix_trace_attr_getclockres()	927						
		posix_trace_attr_getcreatetime()	927						
		posix_trace_attr_getgenversion()	927						
		posix_trace_attr_getinherited()	929						
		posix_trace_attr_getlogfullpolicy()	929						
		posix_trace_attr_getlogsize()	932						

posix_trace_attr_getmaxdatasize()	932
posix_trace_attr_getmaxsystemeventsizes()	932
posix_trace_attr_getmaxusereventsizes()	932
posix_trace_attr_getname()	927, 935
posix_trace_attr_getstreamfullpolicy()	929, 936
posix_trace_attr_getstreamsize()	932, 937
posix_trace_attr_init()	925, 938
posix_trace_attr_setinherited()	929, 939
posix_trace_attr_setlogfullpolicy()	929, 939
posix_trace_attr_setlogsize()	932, 940
posix_trace_attr_setmaxdatasize()	932, 940
posix_trace_attr_setname()	927, 941
posix_trace_attr_setstreamfullpolicy()	929, 942
posix_trace_attr_setstreamsize()	932, 943
posix_trace_clear()	944
posix_trace_close()	946
POSIX_TRACE_CLOSE_FOR_CHILD	929
posix_trace_create()	948
posix_trace_create_withlog()	948
POSIX_TRACE_ERROR trace event	78
posix_trace_event()	952
posix_trace_eventid_equal()	954
posix_trace_eventid_get_name()	954
posix_trace_eventid_open()	952, 956
posix_trace_eventset_add()	957
posix_trace_eventset_del()	957
posix_trace_eventset_empty()	957
posix_trace_eventset_fill()	957
posix_trace_eventset_ismember()	957
posix_trace_eventtypelist_getnext_id()	959
posix_trace_eventtypelist_rewind()	959
posix_trace_event_info structure()	75
POSIX_TRACE_FILTER trace event	78, 963
POSIX_TRACE_FLUSH	930
posix_trace_flush()	948, 960
POSIX_TRACE_FLUSHING	74
POSIX_TRACE_FULL	73-75
posix_trace_getnext_event()	966
posix_trace_get_attr()	961
posix_trace_get_filter()	963
posix_trace_get_status()	961, 965
POSIX_TRACE_INHERITED	929
POSIX_TRACE_LOOP	74, 929-930, 950
POSIX_TRACE_NOT_FLUSHING	74
POSIX_TRACE_NOT_FULL	73-75
POSIX_TRACE_OVERFLOW trace event	78
POSIX_TRACE_OVERRUN	74-75, 961
posix_trace_open()	946, 969
POSIX_TRACE_OVERFLOW trace event	78
POSIX_TRACE_OVERRUN	74-75
POSIX_TRACE_RESUME trace event	78
posix_trace_rewind	946, 969
POSIX_TRACE_RUNNING	73, 972
POSIX_TRACE_SET_EVENTSET	963
posix_trace_set_filter()	963, 970
posix_trace_shutdown()	948, 971
POSIX_TRACE_START trace event	78, 972
posix_trace_start()	972
posix_trace_status_info structure()	73
posix_trace_stop()	972
POSIX_TRACE_STOP trace event	78, 972
POSIX_TRACE_SUB_EVENTSET	963
POSIX_TRACE_SUSPENDED	73-74, 972
POSIX_TRACE_SYSTEM_EVENTS	957
posix_trace_timedgetnext_event()	966, 974
posix_trace_trid_eventid_open()	954, 975
POSIX_TRACE_TRUNCATED_READ	76, 967
POSIX_TRACE_TRUNCATED_RECORD	76, 967
posix_trace_trygetnext_event()	966, 976
POSIX_TRACE_UNTIL_FULL	74, 929-930, 950
POSIX_TRACE_USER_EVENT_MAX	952
POSIX_TRACE_WOPID_EVENTS	957
POSIX_TYPED_MEM_ALLOCATE	785-786883, 977, 979
POSIX_TYPED_MEM_ALLOCATE_CONTIG	785-786, 883, 977, 979
posix_typed_mem_get_info()	977
POSIX_TYPED_MEM_MAP_ALLOCATABLE	832, 979
posix_typed_mem_open()	979
pow()	982
powf()	982
powl()	982
pread()	1190, 985
predefined stream	
standard error	37
standard input	37
standard output	37
preempted thread	1052
PRI	18
printf()	413, 986
priority	38
PRIO_	18
PRIO_INHERIT	1103
PRIO_PGRP	552
PRIO_PROCESS	552
PRIO_USER	552
process	
concurrent execution	404
setting real and effective user IDs	1331
single-threaded	404

process creation	404	pthread_cancel()	1036
process group		PTHREAD_CANCELED	57, 1072
orphaned	316	PTHREAD_CANCEL_ASYNCHRONOUS.....	54
process group ID	548, 1323, 1335	1148
process ID, 1.....	316	PTHREAD_CANCEL_DEFERRED .54, 1050, 1148	
process lifetime	679	PTHREAD_CANCEL_DISABLE	54, 1148
process scheduling	44	PTHREAD_CANCEL_ENABLE.....	54, 1148
process shared memory	1107	pthread_cleanup_pop()	1038
process synchronization	1107	pthread_cleanup_push()	1038
process termination.....	315	pthread_condattr_destroy()	1056
PROT.....	16, 18	pthread_condattr_getclock()	1058
PROT_EXEC.....	786, 795	pthread_condattr_getpshared().....	1060
PROT_NONE.....	44, 785-786, 795	pthread_condattr_init().....	1056, 1062
PROT_READ.....	786, 795	pthread_condattr_setclock()	1058, 1063
PROT_WRITE.....	786-787, 790, 795	pthread_condattr_setpshared().....	1060, 1064
PS.....	7	pthread_cond_broadcast()	1043
pselect()	987	pthread_cond_destroy()	1046
pseudo-random sequence generation.....	1187	pthread_cond_init().....	1046
PST8PDT	1568	PTHREAD_COND_INITIALIZER	1046
ps.....	16	pthread_cond_signal()	1043, 1049
PTHREAD_.....	16	pthread_cond_timedwait()	1050
pthread.....	16	pthread_cond_wait().....	1050
pthread_atfork()	992	pthread_create()	1065
pthread_attr_destroy()	994	PTHREAD_CREATE_DETACHED	30, 997
pthread_attr_getdetachstate()	997	PTHREAD_CREATE_JOINABLE.....	30, 997, 1082
pthread_attr_getguardsize()	999	PTHREAD_DESTRUCTOR_ITERATIONS.....	
pthread_attr_getinheritsched()	1002	1079, 1084, 1494
pthread_attr_getschedparam()	1004	pthread_detach()	1068
pthread_attr_getschedpolicy()	1006	pthread_equal()	1070
pthread_attr_getscope()	1008	pthread_exit()	1071
pthread_attr_getstack()	1010	PTHREAD_EXPLICIT_SCHED	1002
pthread_attr_getstackaddr()	1012	pthread_getconcurrency()	1073
pthread_attr_getstacksize()	1014	pthread_getcpuclockid()	1075
pthread_attr_init()	994, 1016	pthread_getschedparam()	1076
pthread_attr_setdetachstate()	997, 1017	pthread_getspecific()	1079
pthread_attr_setguardsize()	999, 1018	PTHREAD_INHERIT_SCHED	1002
pthread_attr_setinheritsched()	1002, 1019	pthread_join()	1081
pthread_attr_setschedparam()	1004, 1020	PTHREAD_KEYS_MAX	1084, 1494
pthread_attr_setschedpolicy()	1006, 1021	pthread_key_create()	1084
pthread_attr_setscope()	1008, 1022	pthread_key_delete()	1088
pthread_attr_setstack()	1010, 1023	pthread_kill()	1090
pthread_attr_setstackaddr()	1012, 1024	pthread_mutexattr_destroy()	1106
pthread_attr_setstacksize()	1014, 1025	pthread_mutexattr_getprioceiling()	1111
pthread_barrierattr_destroy()	1030	pthread_mutexattr_getprotocol()	1113
pthread_barrierattr_getpshared()	1032	pthread_mutexattr_getpshared()	1115
pthread_barrierattr_init()	1030, 1034	pthread_mutexattr_gettime()	1117
pthread_barrierattr_setpshared()	1032, 1035	pthread_mutexattr_init()	1106, 1119
pthread_barrier_destroy()	1026	pthread_mutexattr_setprioceiling()	1111, 1120
pthread_barrier_init()	1026	pthread_mutexattr_setprotocol()	1113, 1121
PTHREAD_BARRIER_SERIAL_THREAD	1028	pthread_mutexattr_setpshared()	1115, 1122
pthread_barrier_wait()	1028	pthread_mutexattr_settype()	1117, 1123

PTHREAD_MUTEX_DEFAULT	1099, 1117
pthread_mutex_destroy()	1091
PTHREAD_MUTEX_ERRORCHECK	1099, 1117
pthread_mutex_getprioceiling().....	1096
pthread_mutex_init()	1091, 1098
PTHREAD_MUTEX_INITIALIZER	1091, 1098
pthread_mutex_lock()	1099
PTHREAD_MUTEX_NORMAL	1099, 1117
PTHREAD_MUTEX_RECURSIVE	10991117-1118
pthread_mutex_setprioceiling().....	1096, 1102
pthread_mutex_timedlock().....	1103
pthread_mutex_trylock().....	1099, 1105
pthread_mutex_unlock()	1099, 1105
pthread_once()	1124
PTHREAD_ONCE_INIT	1124
PTHREAD_PRIO_INHERIT	1113
PTHREAD_PRIO_NONE	1113
PTHREAD_PRIO_PROTECT	1100, 1113
PTHREAD_PROCESS_PRIVATE	1032, 10601107, 1115, 1143, 1159
PTHREAD_PROCESS_SHARED	1032, 10601107, 1115, 1143, 1159
pthread_rwlockattr_destroy().....	1141
pthread_rwlockattr_getpshared()	1143
pthread_rwlockattr_init()	1141, 1145
pthread_rwlockattr_setpshared()	1143, 1146
pthread_rwlock_destroy()	1126
pthread_rwlock_init()	1126
pthread_rwlock_rdlock()	1128
pthread_rwlock_timedrdlock()	1131
pthread_rwlock_timedwrlock()	1133
pthread_rwlock_tryrdlock()	1128, 1135
pthread_rwlock_trywrlock()	1136
pthread_rwlock_unlock()	1138
pthread_rwlock_wrlock()	1136, 1140
PTHREAD_SCOPE_PROCESS	52, 1008
PTHREAD_SCOPE_SYSTEM	52, 1008
pthread_self()	1147
pthread_setcancelstate()	1148
pthread_setcanceltype(0	1148
pthread_setconcurrency()	1073, 1150
pthread_setschedparam()	1076, 1151
pthread_setschedprio()	1152
pthread_setspecific()	1079, 1154
pthread_sigmask()	1155
pthread_spin_destroy()	1159
pthread_spin_init(0.....	1159
pthread_spin_lock()	1161
pthread_spin_trylock(0	1161
pthread_spin_unlock()	1163
PTHREAD_STACK_MIN	1010, 1012, 1014, 1494
pthread_testcancel()	1148, 1164
PTHREAD_THREADS_MAX	1065, 1494
ptsname()	1165
putc()	1166
putchar()	1168
putchar_unlocked()	496, 1169
putc_unlocked()	496, 1167
putenv()	1170
putmsg()	1172
putpmsg	1172
puts()	1176
pututxline()	289, 1178
putwc()	1179
putwchar()	1180
pwrite()	1676, 1181
pw_	16
p_	16
P_	17
P_ALL	1615
P_PGID	1615
P_PID	1615
qsort()	1182
queue a signal to a process	1389
raise()	1184
rand()	1186
random()	617, 1189
RAND_MAX	1186
rand_r()	1186
read from a file	1193
read()	1190
readdir()	1197
readdir_r	1197
readlink()	1201
readv()	1204
real user ID	100, 678
realloc()	1206
realpath()	1208
REALTIME	107, 109-110, 112, 115-116118, 202, 208, 345, 695, 781783, 797-798, 800, 802, 805, 808810, 814, 835, 1251-1255, 12581266-1268, 1270, 1272, 1275, 12791281, 1283, 1347, 1351, 1388, 13951401, 1540, 1543-1544
REALTIME THREADS	1002, 1006, 10081019, 1021-1022, 1076, 1096, 11021111, 1113, 1120-1121, 1151-1152
recv()	1210
recvfrom()	1212
recvmsg()	1215

regcomp()	1218	RLIMIT_FSIZE	563
reerror().....	1218	RLIMIT_NOFILE.....	563, 565
regexec().....	1218	RLIMIT_STACK.....	563
regfree().....	1218	rlim_	16
register fork handlers.....	992	RLIM_.....	18
REG_.....	18	RLIM_INFINITY	563-564
REG_ constants		RLIM_SAVED_CUR.....	564
error return values of regcomp	1220	RLIM_SAVED_MAX.....	564
used in regcomp	1218	rmdir().....	1241
REG_BADBR	1220	RMSGD.....	625
REG_BADPAT	1220	RMSGN.....	625
REG_BADRPT	1220	RNORM.....	625
REG_EBRACE	1220	round robin	46
REG_EBRACK	1220	round()	1244
REG_ECOLLATE	1220	roundf()	1244
REG_ECTYPE	1220	roundl()	1244
REG_EESCAPE	1220	routing	60
REG_EPAREN	1220	RPROTDAT.....	625
REG_ERANGE	1220	RPROTDIS	625
REG_ESPACE	1220	RPROTNORM	625
REG_ESUBREG	1220	RS	7
REG_EXTENDED	1218	RS_HIPRI	533, 624, 1172
REG_ICASE	1218	RTLD_	18
REG_NEWLINE	1218	RTLD_DEFAULT	266
REG_NOMATCH	1220	RTLD_GLOBAL	259, 263-264, 267
REG_NOSUB	1218	RTLD_LAZY	263, 266
REG_NOTBOL	1219	RTLD_LOCAL	264
REG_NOTEOL	1219	RTLD_NEXT	266-267
remainder()	1225	RTLD_NOW	263-264
remainderf()	1225	RTS.....	7
remainderl()	1225	RTSIG_MAX	1494
remove a directory	1242	RUSAGE_	18
remove directory entries	1583	RUSAGE_CHILDREN	566
remove().....	1227	RUSAGE_SELF	566
remque()	619, 1229	ru_	16
remquo()	1230	s6_	16
remquof()	1230	sa_	16
remquol()	1230	SA_	18
rename a file	1234	SA_NOCLDSTOP	31, 1362-1363, 1366-1367
rename().....	1232	SA_NOCLDWAIT	313-314, 566, 1364, 1608
rewind()	1236	SA_NODEFER	1364
rewinddir()	1237	SA_ONSTACK	303, 1363
re_.....	16	SA_RESETHAND	153, 1363-1364
RE_DUP_MAX	1494	SA_RESTART	153, 990, 1363, 1378
rindex().....	1238	SA_SIGINFO	1362-1363, 1366, 1388
rint()	1239	scalb()	1246
RLIMIT_	18	scalbln()	1248
RLIMIT_AS	564	scalblnf()	1248
RLIMIT_CORE	563	scalblnl()	1248
RLIMIT_CPU	563	scalbn()	1248
RLIMIT_DATA	563	scalbnf()	1248

scalbnl().....	1248
scanf().....	444, 1250
schedule alarm	121
scheduling documentation	54
scheduling policy	
round robin.....	46
SCHED_.....	16
sched_.....	16
SCHED_FIFO	42, 45, 53, 303, 402
.....	552, 844, 1004, 1006, 1076, 1111
.....	1128, 1256, 1275
sched_getparam()	1252
sched_getscheduler()	1253
sched_get_priority_max()	1251
sched_get_priority_min()	1251
SCHED_OTHER	45, 48, 552, 1006, 1076, 1256
SCHED_RR	42, 45-46, 53, 303, 402, 552
.....	844, 1004, 1006, 1076, 1128, 1256, 1275
sched_rr_get_interval()	1254
sched_setparam().....	1255
sched_setscheduler().....	1258
SCHED_SPORADIC	42, 45-46, 1128, 1256, 1275
sched_yield().....	1261
SCM_.....	18
SCN.....	18
SD.....	7
security considerations	196, 315, 678, 1323, 1425
seed48().....	268, 1262
seekdir()	1263
SEEK_CUR	337, 451, 738
SEEK_END	337, 451, 738
SEEK_GET	1236
SEEK_SET	42, 112, 118, 337, 451, 738
SEGV_.....	18
select().....	987, 1265
SEM.....	8
semctl().....	1284
semget()	1287
semid	40
semop()	1290
SEM_.....	16, 18
sem_.....	16
sem_close()	1266
sem_destroy()	1267
SEM_FAILED	1273
sem_getvalue()	1268
sem_init()	1270
SEM_NSEMS_MAX.....	1270, 1494
sem_open()	1272
sem_perm.....	40
sem_post()	1275
sem_timedwait()	1277
sem_trywait()	1279
SEM_UNDO	1290
sem_unlink()	1281
SEM_VALUE_MAX.....	1272, 1494
sem_wait()	1279, 1283
send()	1295
sendmsg()	1297
sendto()	1300
service name	435
session	316, 678, 1323, 1335
set cancelability state	1148
set file creation mask	1574
set process group ID for job control	1323
set-group-ID	194, 309, 315, 342
set-user-ID	309, 315, 503, 678
SETALL	1284, 1287
setbuf()	1303
setcontext()	500, 1304
setegid()	1305
setenv()	1306
seteuid()	1308
setgid()	1309
setgrent()	277, 1311
sethostent()	279, 1312
setitimer()	528, 1313
setjmp()	1314
setkey()	1316
setlocale()	1317
setlogmask()	215, 1321
setnetent()	281, 1322
setpgid()	1323
setpgrp()	1325
setpriority()	552, 1326
setprotoent()	283, 1327
setpwent()	285, 1328
setregid()	1329
setreuid()	1331
setrlimit()	563, 1333
setservent()	287, 1334
setsid()	1335
setsockopt()	1337
setstate()	617, 1340
setuid()	1341
setutxent()	289, 1344
SETVAL	1284, 1287
setvbuf()	1345
shall.....	2
shell.....	308, 316, 531, 548, 678, 1324, 1612
job	678
login.....	531

shell scripts	1382
exec	308
shell, login	308
SHM	8, 18
shmat()	1353
shmctl()	1355
shmdt()	1357
shmget()	1358
shmid	40
SHMLBA	1353
shm_	16
SHM_	18
shm_open()	1347
shm_perm	40
SHM_RDONLY	1353
SHM_RND	1353
shm_unlink()	1351
should	2
shutdown()	1360
SHUT_	18
SIGABRT	94
sigaction()	1362
sigaddset()	1369
SIGALRM	121, 528, 1407, 1570, 1587
sigaltstack()	1370
SIGBUS	44, 787, 790, 1155
SIGCANCEL	1036
SIGCHLD	216, 313-314, 566, 5911363, 1366, 1376, 1500, 1608, 1615
SIGCLD	1366
SIGCONT	34, 314, 316, 677-678
sigdelset()	1372
sigemptyset()	1373
SIGEV_	16
sigev_	16
SIGEV_NONE	29, 42
SIGEV_SIGNAL	29, 1540
SIGEV_THREAD	29-30
sigfillset()	1375
SIGFPE	1155, 1382
sighold()	1376
SIGHUP	210, 314, 316
sigignore()	1376
SIGILL	1155, 1382
SIGINT	404, 1500
siginterrupt()	1378
sigismember()	1380
SIGKILL	678, 1362, 1366-1367, 1376
siglongjmp()	1381
signal generation and delivery	28
realtime	29
signal handler	1382
signal()	1382
signaling a condition	1044
signals	28
signbit()	1384
sigpause()	1376, 1385
sigpending()	1386
SIGPIPE	334, 368, 425, 429, 452, 455, 1173, 1679
SIGPOLL	210, 623-624
sigprocmask()	1155, 1387
SIGPROF	528
sigqueue()	1388
SIGQUEUE_MAX	1388, 1494
SIGQUIT	1500
sigrelse()	1376, 1390
SIGRTMAX	29-30, 1388, 1395, 1399
SIGRTMIN	29-30, 1388, 1395, 1399
SIGSEGV	44, 564, 832, 999, 1155, 1382
sigset	1376, 1390
sigsetjmp()	1391
SIGSTKSZ	1370
SIGSTOP	29, 1362, 1367, 1376
sigsuspend()	1393
sigtimedwait()	1395
SIGTSTP	29
SIGTTIN	29, 372, 378, 1192
SIGTTOU	29, 334, 368, 425, 429452, 454, 1509, 1511, 1513, 15201523, 1679
SIGURG	624
SIGVTALRM	528
sigwait()	1399
sigwaitinfo()	1395, 1401
SIGXCPU	563
SIGXFSZ	563, 1561
SIG_	16, 18
SIG_BLOCK	1155
SIG_DFL	30, 302, 564, 1362, 1364, 1382
SIG_ERR	153, 1382
SIG_HOLD	1376
SIG_IGN	30-31, 302, 309, 313-314566, 1362, 1382, 1608
SIG_SETMASK	1155
SIG_UNBLOCK	1155
sin()	1402
sin6_	16
sinf()	1402
sinh()	1404
sinhf()	1404
sinhl()	1404
sinl()	1402, 1406

sin.....	16	sqrt()	1417
SIO.....	8	sqrtf()	1417
SIOCATMARK.....	1410	sqrtl()	1417
sival_.....	16	rand()	1186, 1419
SI_.....	16, 18	rand48()	268, 1420
si_.....	16-17	random()	617, 1421
SI_ASYNCIO.....	32	SS.....	8
SI_MESGQ.....	32	sscanf()	444, 1422
SI_QUEUE.....	32	SSIZE_MAX.....	805, 821, 1190, 1201, 1444, 1676
SI_TIMER.....	32	ss_.....	16-17
SI_USER.....	32	SS_.....	18
sleep()	1407	SS_DISABLE.....	1370-1371
sl_.....	16	SS_ONSTACK.....	1370
SND.....	18	stack size.....	994
SNDZERO.....	627	stat()	1423
snprintf().....	413, 1409	statvfs()	458, 1427
SO.....	18	stderr	1428
socketmark().....	1410	STDERR_FILENO.....	1428
socket I/O mode.....	61	stdin.....	1428
socket out-of-band data	62	STDIN_FILENO.....	874, 1428
socket owner.....	61	stdio locking functions	383
socket queue limits.....	61	stdio with explicit client locking	496
socket receive queue	61	stdout	1428
socket types.....	60	STDOUT_FILENO.....	874, 1428
socket().....	1412	STR.....	18
socketpair().....	1414	strcasecmp()	1430
sockets.....	59	strcat()	1431
address families	59	strchr()	1432
addressing.....	59	strcmp()	1433
asynchronous errors	63	strcoll()	1435
connection indication queue.....	62	strcpy()	1437
Internet Protocols	67	strcspn()	1439
IPv4.....	68	strupr()	1440
IPv6.....	68	STREAM	626, 628, 1172, 1191, 1678
local UNIX connections	67	stream	
options	64	byte-oriented	36
pending error.....	61	wide-oriented	36
protocols.....	60	STREAM head/tail.....	38
signals	63	stream-full-policy attribute	73-74, 77, 930
SOCK.....	18	stream-min-size attribute	77, 933
SOCK_DGRAM.....	67, 1412, 1414	streams	34
SOCK_RAW.....	67	STREAMS	22, 210, 326, 346, 533
SOCK_SEQPACKET.....	67, 1412, 1414	622, 637, 852-853, 870, 987
SOCK_STREAM.....	67, 1412, 1414	access	39
SPI.....	8	multiplexed.....	630
SPN.....	8	streams	
sporadic server policy		interaction with file descriptors	35
execution capacity.....	46	overview	38
replenishment period	46	stream orientation	36
sprintf()	413, 1416	STREAM_MAX.....	350, 399, 874, 1494
spurious wakeup.....	1044		

strerror().....	1441	synchronously accept a signal.....	1396
strerror_r().....	1441	sysconf().....	1492
strfmon().....	1443	syslog().....	215, 1499
strftime().....	1447	system crash	461
strlen()	1453	System III	196, 1576
strncasecmp()	1430, 1455	system interfaces	85
strncat()	1456	system name.....	1576
strncmp()	1457	system trace event type definitions	77
strncpy().....	1458	System V.....	121, 196, 310, 316, 341
strpbrk().....	1459	409, 548, 678, 767, 1242, 1335
strptime().....	1460	1366-1367, 1391, 1515, 1576
strrchr()	1464	system().....	1500
strspn()	1465	S_	16
strstr()	1466	S_.....	18
strtod().....	1467	S_BANDURG	624
strtof().....	1467	S_ERROR.....	623
strtoimax().....	1471	S_HANGUP.....	624
strtok().....	1472	S_HIPRI	623
strtok_r().....	1472	S_IFBLK	772
strtol().....	1475	S_IFCHR	772
strtold().....	1467, 1477	S_IFDIR	772
strtoll()	1475, 1478	S_IFIFO	772
strtoul()	1479	S_IFREG	772
strtoull()	1479	S_INPUT.....	623
strtoumax().....	1471, 1482	S_IRGRP.....	330, 456, 772
strxfrm().....	1483	S_IROTH.....	330, 456, 772
str_.....	16	S_IRUSR.....	330, 456, 772
st_.....	16	S_IRWXG	772
ST_.....	18	S_IRWXO	772
ST_NOSUID	303, 458	S_IRWXU	772
ST_RDONLY	458	S_ISgid.....	192-194, 772, 1561, 1677
sun_	17	S_ISuid.....	192-193, 772, 1561, 1677
superuser	100, 196, 694, 1583	S_ISVTX	192, 772, 1233, 1242, 1581
supplementary groups.....	196, 521	S_IWGRP	330, 456, 772
SVID	1391	S_IWOTH	330, 456, 772
SVR4.....	789, 835	S_IWUSR	330, 456, 772
sv_.....	16	S_IXGRP	772
SV_.....	18	S_IXOTH	772
swab().....	1485	S_IXUSR	772
swapcontext().....	742, 1486	S_MSG.....	623
swprintf()	477, 1487	S_OUTPUT	623
swscanf()	486, 1488	S_RDBAND.....	623-624
SWTCH.....	19	S_RDNORM	623
symbols		S_WRBAND	623
POSIX.1.....	13	S_WRNORM	623
symlink().....	1489	TABSIZE	155, 736
SYMLINK_MAX	407, 1489	tan().....	1504
SYMLOOP_MAX	152, 225, 327, 346, 399	tanf()	1504
.....	440, 459, 467, 473, 683, 773, 842	tanh()	1506
.....	1208, 1299, 1302, 1494, 1562, 1591	tanhf()	1506
sync()	1491	tanhl()	1506

tanl()	1504, 1508	timer	17
tcdrain()	1509	TIMER_ABSTIME	49, 205, 1544
tcflow()	1511	timer_create()	1540
tcflush()	1513	timer_delete()	1543
tcgetattr()	1515	timer_getoverrun()	1544
tcgetpgrp()	1517	timer_gettime()	1544
tcgetsid()	1519	TIMER_MAX	1494
TCIFLUSH	1513	timer_settime()	1544
TCIOFF	1511	times()	1547
TCIOFLUSH	1513	timezone()	1549
TCION	1511	TMO	9
TCOFLUSH	1513	tmpfile()	1550
TCOOFF	1511	tmpnam()	1552
TCOON	1511	TMP_MAX	1532, 1551-1552
TCP_	18	TMR	9
TCSADRAIN	1522	tms_	16
TCSAFLUSH	1522	tm_	17
TCSANOW	1522	toascii()	1554
tcsendbreak()	1520	tolower()	1555
tcsetattr()	1522	TOSTOP	334, 368, 425, 429, 452, 454, 1678
tcsetpgrp()	1525	toupper()	1556
TCT	8	towctrans()	1557
tdelete()	1527	towlower()	1558
TEF	8	towupper()	1559
telldir()	1531	TPI	9
tempnam()	1532	TPP	9
terminal access control	1515, 1523	TPS	9
terminal device name	1565	trace event, POSIX_TRACE_ERROR	78
terminate a process	315	trace event, POSIX_TRACE_FILTER	78, 963
terminology	1	trace event, POSIX_TRACE_OVERFLOW	78
termios structure	1515	trace event, POSIX_TRACE_RESUME	78
tfind()	1527, 1534	trace event, POSIX_TRACE_START	78, 972
tgamma()	1535	trace event, POSIX_TRACE_STOP	78, 972
tgammaf()	1535	trace functions	80
tgammal()	1535	trace-name attribute	77, 927
THR	9	TRACE_EVENT_NAME_MAX	952, 954
thread creation	1066	TRACE_SYS_MAX	949
thread creation attributes	994	TRACE_USER_EVENT_MAX	952, 954
thread ID	51, 1070	TRACING	925, 927, 929, 932, 935-944
thread mutexes	51	946, 948, 952, 954, 956-957, 959-961
thread scheduling	52	963, 965-966, 969-972, 974-976
thread termination	1071	TRAP_	18
thread-safety	50, 383	TRC	9
thread-specific data key creation	1086	TRI	10
thread-specific data key deletion	1088	TRL	10
thread-specific data management	1080	trunc()	1560
threads	50	truncate()	1561
regular file operations	58	truncation-status attribute	952
time()	1537	truncf()	1560, 1563
timer ID	1542	truncl()	1560, 1563
TIMER_	17, 18	TSA	10

tsearch()	1527, 1564	va_copy()	1593
TSF	10	va_end()	1593
TSH	10	va_start()	1593
TSP	10	VDISCARD	19
TSS	10	VDSUSP	19
ttynname()	1565	Version 7	121, 196, 678, 1576
ttynname_r()	1565	vfork()	1594
TTY_NAME_MAX	1494, 1565	vfprintf()	1596
tv_	16-17	vfscanf()	1597
twalk()	1527, 1567	vfwprintf()	1598
TYM	10	vfwscanf()	1599
tzname	1568	VISIT	1527, 1567
TZNAME_MAX	1494	VLNEXT	19
tzset()	1568, 1568	vprintf()	1596, 1600
t_uscalar_t	625	VREPRINT	19
ualarm()	1570	vscanf()	1597, 1601
uc_	16-17	vsnprintf()	1596, 1602
UINT_MAX	121, 1408	vsprintf()	1596, 1602
UIO_MAXIOV	17	vsscanf()	1597, 1603
ulimit()	1572	VSTATUS	19
ULLONG_MAX	1480	vswprintf()	1598, 1604
ULONG_MAX	1480, 1652	vswscanf()	1599, 1605
UL_	17	VWERASE	19
UL_GETFSIZE	1572	vwprintf()	1598, 1606
UL_SETFSIZE	1572	vwscanf()	1599, 1607
umask()	1574	wait for process termination	1612
uname()	1576	wait for thread termination	1082
undefined	2	wait()	1608
underlying function	36	waitid()	1615
ungetc()	1578	waiting on a condition	1051
ungetwc()	1579	waitpid()	1608, 1617
unicast	68	WARNING	394
unlink()	1581	warning	
unlockpt()	1585	OB	6
unsetenv()	1586	OF	7
unspecified	2	WCONTINUED	1608, 1615
UP	11	wcrtomb()	1618
US-ASCII	636	wcscat()	1620
user ID		wcschr()	1621
real and effective	1331	wcscmp()	1622
setting real and effective	1331	wcscoll()	1623
user trace event type definitions	80	wcscopy()	1624
USER_PROCESS	289-290	wcscspn()	1625
usleep()	1587	wcsftime()	1626
UTC	1568	wcslen()	1628
utime()	1589	wcsncat()	1629
utimes()	1591	wcsncmp()	1630
utim_	17	wcsncpy()	1631
uts_	17	wcspbrk()	1632
ut_	17	wcsrchr()	1633
va_arg()	1593	wcsrtombs()	1634

wcsspn().....	1636	WRDE_UNDEF.....	1672
wcsstr().....	1637	write to a file.....	1680
wcstod()	1638	write().....	1676
wcstof()	1638	writev()	1684
wcstoiimax()	1641	wscanf()	486, 1686
wcstok()	1642	WSTOPPED	1615
wcstol()	1644	WSTOPSIG	1609
wcstold().....	1638, 1647	WTERMSIG	1609
wcstoll().....	1644, 1648	WUNTRACED	1608, 1612
wcstombs().....	1649	XSI.....	11
wcstoul().....	1651	XSI interprocess communication	39
wcstoumax().....	1641, 1654	XSR	11
wcswcs().....	1655	X_OK	100
wcswidth().....	1656	y0().....	1687
wcsxfrm().....	1657	y1().....	1687
wctob()	1659	yn().....	1687
wctomb()	1660	zombie process.....	313
wctrans().....	1662		
wctype().....	1663		
wcwidth()	1665		
WEOF.....	83, 659-660, 662-663, 665-672		
 1558-1559, 1579		
WEXITED.....	1615		
WEXITSTATUS	1609		
we.....	17		
wide-oriented stream.....	36		
WIFCONTINUED	1609		
WIFEXITED	1609		
WIFSIGNALLED	1609		
WIFSTOPPED	1609, 1613		
wmemchr().....	1666		
wmemcmp().....	1667		
wmemcpy().....	1668		
wmemmove()	1669		
wmemset()	1670		
WNOHANG.....	1367, 1608, 1615		
WNOWAIT.....	1615		
wordexp().....	1671		
wordfree().....	1671		
wprintf()	477, 1675		
WRDE	18		
WRDE_APPEND	1672		
WRDE_BADCHAR	1673		
WRDE_BADVAL	1673		
WRDE_CMDSUB	1673		
WRDE_DOOFFS.....	1672		
WRDE_NOCMD	1672		
WRDE_NOSPACE	1673		
WRDE_REUSE	1672		
WRDE_SHOWERR	1672		
WRDE_SYNTAX	1673		