IEEE Std 1003.1-2001
(Revision of IEEE Std 1003.1-1996
and IEEE Std 1003.2-1992)
Open Group Technical Standard
Base Specifications, Issue 6

1003.1™
Standard for Information Technology —
Portable Operating System Interface (POSIX®)
System Interfaces, Issue 6

Approved 12 September 2001
The Open Group

IEEE Sponsor
Portable Applications Standards Committee
of the
IEEE Computer Society

Approved 6 December 2001
IEEE-SA Standards Board
Abstract

This standard defines a standard operating system interface and environment, including a command interpreter (or “shell”), and common utility programs to support applications portability at the source code level. It is the single common revision to IEEE Std 1003.1-1996, IEEE Std 1003.2-1992, and the Base Specifications of The Open Group Single UNIX® Specification, Version 2. This standard is intended to be used by both applications developers and system implementors and comprises four major components (each in an associated volume):

• General terms, concepts, and interfaces common to all volumes of this standard, including utility conventions and C-language header definitions, are included in the Base Definitions volume.

• Definitions for system service functions and subroutines, language-specific system services for the C programming language, function issues, including portability, error handling, and error recovery, are included in the System Interfaces volume.

• Definitions for a standard source code-level interface to command interpretation services (a “shell”) and common utility programs for application programs are included in the Shell and Utilities volume.

• Extended rationale that did not fit well into the rest of the document structure, containing historical information concerning the contents of this standard and why features were included or discarded by the standard developers, is included in the Rationale (Informative) volume.

The following areas are outside the scope of this standard:

• Graphics interfaces

• Database management system interfaces

• Record I/O considerations

• Object or binary code portability

• System configuration and resource availability

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

Keywords

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

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

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

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

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

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

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of the IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

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

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

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

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

The IEEE and its designees are the sole entities that may authorize the use of IEEE-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein. Authorization to photocopy portions of any individual standard for internal or personal use is granted in the U.S. by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to the Copyright Clearance Center. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center. To arrange for payment of the licensing fee, please contact:

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

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

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

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

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

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.

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

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

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

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

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

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

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

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

- A new Version indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it replaces the previous publication.

- A new Issue indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

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

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

### Chapter 1 Introduction
- 1.1 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
- 2.1 Use and Implementation of Functions .................................................. 13
- 2.2 The Compilation Environment ............................................................... 13
- 2.2.1 POSIX.1 Symbols .................................................................................. 13
- 2.2.1.1 The _POSIX_C_SOURCE Feature Test Macro .................................. 14
- 2.2.1.2 The _XOPEN_SOURCE Feature Test Macro ..................................... 14
- 2.2.2 The Name Space .................................................................................... 14
- 2.3 Error Numbers.......................................................................................... 21
- 2.3.1 Additional Error Numbers..................................................................... 28
- 2.4 Signal Concepts........................................................................................ 28
- 2.4.1 Signal Generation and Delivery ............................................................ 28
- 2.4.2 Realtime Signal Generation and Delivery ............................................. 29
- 2.4.3 Signal Actions........................................................................................ 31
- 2.4.4 Signal Effects on Other Functions ......................................................... 34
- 2.5 Standard I/O Streams............................................................................ 34
- 2.5.1 Interaction of File Descriptors and Standard I/O Streams ................. 35
- 2.5.2 Stream Orientation and Encoding Rules .............................................. 37
- 2.6 STREAMS ................................................................................................ 38
- 2.6.1 Accessing STREAMS ......................................................................... 39
- 2.7 XSI Interprocess Communication ............................................................ 39
- 2.7.1 IPC General Description ..................................................................... 40
- 2.8 Realtime .................................................................................................... 41
- 2.8.1 Realtime Signals...................................................................................... 41
- 2.8.2 Asynchronous I/O ................................................................................. 41
- 2.8.3 Memory Management .......................................................................... 43
- 2.8.3.1 Memory Locking................................................................................ 43
- 2.8.3.2 Memory Mapped Files....................................................................... 43
- 2.8.3.3 Memory Protection............................................................................ 44
- 2.8.3.4 Typed Memory Objects ................................................................... 44
- 2.8.4 Process Scheduling .............................................................................. 44
Contents

2.11.3 Trace Functions .......................................................... 79
2.12 Data Types ................................................................. 80

Chapter 3 System Interfaces ............................................. 83

Index .................................................................................. 1663

List of Tables

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

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

This standard has been jointly developed by the IEEE and The Open Group. It is both an IEEE Standard and an Open Group Technical Standard.

The Austin Group

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

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

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

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

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

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

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

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

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

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

Audience

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

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

Purpose

Several principles guided the development of this standard:

- Application-Oriented

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

- Interface, Not Implementation

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

---

\(^4\) The name POSIX was suggested by Richard Stallman. It is expected to be pronounced \textit{paeh-icks}, as in \textit{positive}, not \textit{poh-six}, or other variations. The pronunciation has been published in an attempt to promulgate a standardized way of referring to a standard operating system interface.
• Source, Not Object, Portability

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

• The C Language

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

• No Superuser, No System Administration

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

• Minimal Interface, Minimally Defined

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

• Broadly Implementable

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

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

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

• Minimal Changes to Historical Implementations

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

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

1. By standardizing an interface like one in an historical implementation; for example, directories

2. By specifying an interface that is readily implementable in terms of, and backwards-compatible with, historical implementations, such as the extended `tar` format defined in the `pax` utility

3. By specifying an interface that, when added to an historical implementation, will not conflict with it; for example, the `sigaction()` function

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

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

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

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

• Minimal Changes to Existing Application Code

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

This Standard

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

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

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

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

**Notes:**

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

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

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

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

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

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

**Notes:**

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

The Austin Group

This standard was prepared by the Austin Group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society, The Open Group, and ISO/SC22 WG15. At the time of approval, the membership of the Austin Group was as follows.

Andrew Josey, Chair
Donald W. Cragun, Organizational Representative, IEEE PASC
Nicholas Stoughton, Organizational Representative, ISO/SC22 WG15
Mark Brown, Organizational Representative, The Open Group
Cathy Hughes, Technical Editor

Austin Group Technical Reviewers

Peter Anvin           Michael Gonzalez           Sandra O'Donnell
Bouazza Bachar       Joseph M. Gwinn            Frank Prindle
Walter Briscoe        Yvette Ho Sang            Glen Seeds
Mark Brown            Cathy Hughes              Keld Jorn Simonsen
Dave Butenhof         Lowell G. Johnson         Raja Srinivasan
Geoff Clare           Andrew Josey              Nicholas Stoughton
Donald W. Cragun      Michael Kavanaugh        Donn S. Terry
Lee Damico            David Korn                Fred Tydeman
Ulrich Drepper        Marc Aurele La France     Peter Van Der Veen
Paul Eggert           Jim Meyering              James Youngman
Joanna Farley          Gary Miller              Jim Zepeda
Clive D.W. Feather    Finnbarr P. Murphy        Jason Zions
Andrew Gollan          Joseph S. Myers
Austin Group Working Group Members

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

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

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        James H. Gurney        James W. Moore
Mark D. Bowman           Richard J. Holleman     Robert F. Munzner
Gary R. Engmann          Lowell G. Johnson      Ronald C. Petersen
Harold E. Epstein        Robert J. Kennelly    Gerald H. Peterson
H. Landis Floyd          Joseph L. Koepfinger*  John B. Posey
Jay Forster*             Peter H. Lips          Gary S. Robinson
Howard M. Frazier        L. Bruce McClung      Akio Tojo
Ruben D. Garzon          Daleep C. Mohla       Donald W. Zipse

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

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

* Member Emeritus
Trademarks

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

1003.1™ is a trademark of the Institute of Electrical and Electronic Engineers, Inc.
AIX® is a registered trademark of IBM Corporation.
AT&T® is a registered trademark of AT&T in the U.S.A. and other countries.
BSD™ is a trademark of the University of California, Berkeley, U.S.A.
Hewlett-Packard®, HP®, and HP-UX® are registered trademarks of Hewlett-Packard Company.
IBM® is a registered trademark of International Business Machines Corporation.
Motif®, OSF/1®, UNIX®, and the “X Device” are registered trademarks and IT DialTone™ and The Open Group™ are trademarks of The Open Group in the U.S. and other countries.
POSIX® is a registered trademark of the Institute of Electrical and Electronic Engineers, Inc.
Sun® and Sun Microsystems® are registered trademarks of Sun Microsystems, Inc.
/usr/group® is a registered trademark of UniForum, the International Network of UNIX System Users.
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.
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

Almasi and Gottlieb

ANSI C

ANSI X3.226-1994

Brawer

DeRemer and Pennello Article

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

IEC 60559: 1989

IEEE Std 754-1985

IEEE Std 854-1987
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

IETF RFC 819

IETF RFC 822

IETF RFC 919
Broadcasting Internet Datagrams, J. Mogul, October 1984.

IETF RFC 920

IETF RFC 921
Domain Name System Implementation Schedule, J. Postel, October 1984.

IETF RFC 922

IETF RFC 1034

IETF RFC 1035

IETF RFC 1123
Requirements for Internet Hosts — Application and Support, R. Braden, October 1989.

IETF RFC 1886

IETF RFC 2045
Multipurpose Internet Mail Extensions (MIME), Part 1: Format of Internet Message Bodies, N. Freed, N. Borenstein, November 1996.

IETF RFC 2373

IETF RFC 2460

Internationalisation Guide

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 8652: 1987

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 6429: 1992

ISO/IEC 6937: 1994

ISO/IEC 8802-3: 1996

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

Part 1: Latin Alphabet No. 1
Part 2: Latin Alphabet No. 2
Part 3: Latin Alphabet No. 3
Part 4: Latin Alphabet No. 4
Part 5: Latin/Cyrillic Alphabet
Part 6: Latin/Arabic Alphabet
Part 7: Latin/Greek Alphabet
Part 8: Latin/Hebrew Alphabet
Part 9: Latin Alphabet No. 5
Part 10: Latin Alphabet No. 6
Part 13: Latin Alphabet No. 7
Part 14: Latin Alphabet No. 8
Part 15: Latin Alphabet No. 9

ISO POSIX-1: 1996

ISO POSIX-2: 1993

Issue 1
Referenced Documents

Issue 2
X/Open Portability Guide, January 1987:
• Volume 1: XVS Commands and Utilities (ISBN: 0-444-70174-5)

Issue 3

Issue 4
CAE Specification, July 1992, published by The Open Group:

Issue 4, Version 2
CAE Specification, August 1994, published by The Open Group:

Issue 5
Technical Standard, February 1997, published by The Open Group:
• System Interface Definitions (XBD), Issue 5 (ISBN: 1-85912-186-1, C605)
• 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
Referenced Documents

MSE Working Draft

POSIX.0: 1995

POSIX.1: 1988

POSIX.1: 1990

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

POSIX.1g: 2000

POSIX.1j: 2000

POSIX.1q: 2000

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

POSIX.2d:-1994

POSIX.13:-1998
Referenced Documents

Sarwate Article

Sprunt, Sha, and Lehoczky

SVID, Issue 1

SVID, Issue 2

SVID, Issue 3

The AWK Programming Language

UNIX Programmer’s Manual

XNS, Issue 4

XNS, Issue 5

XNS, Issue 5.2

X/Open Curses, Issue 4, Version 2

Yacc

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

AIX 3.2 Manual

OSF/1
Referenced Documents

OSF AES

System V Release 2.0

System V Release 4.2
1.1 Scope


1.2 Conformance


1.3 Normative References


1.4 Change History

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

1.5 Terminology

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

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

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

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

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

legacy
Describes a feature or behavior that is being retained for compatibility with older applications, but which has limitations which make it inappropriate for developing portable
applications. New applications should use alternative means of obtaining equivalent functionality.

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

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

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

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

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

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

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

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

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

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


1.7 Relationship to Other Formal Standards

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

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

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

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

1.8 Portability

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

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

1.8.1 Codes

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

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

Where applicable, functions are marked with the ADV margin legend in the SYNOPSIS section.

Where additional semantics apply to a function, the material is identified by use of the ADV margin legend.

AIO Asynchronous Input and Output
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the AIO margin legend in the SYNOPSIS section.

Where additional semantics apply to a function, the material is identified by use of the AIO margin legend.

BAR Barriers
The functionality described is optional. The functionality described is also an extension to the
ISO C standard.
Where applicable, functions are marked with the BAR margin legend in the SYNOPSIS section.
Where additional semantics apply to a function, the material is identified by use of the BAR margin legend.

Batch Environment Services and Utilities
The functionality described is optional.
Where applicable, utilities are marked with the BE margin legend in the SYNOPSIS section.
Where additional semantics apply to a utility, the material is identified by use of the BE margin legend.

C-Language Development Utilities
The functionality described is optional.
Where applicable, utilities are marked with the CD margin legend in the SYNOPSIS section.
Where additional semantics apply to a utility, the material is identified by use of the CD margin legend.

Process CPU-Time Clocks
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the CPT margin legend in the SYNOPSIS section.
Where additional semantics apply to a function, the material is identified by use of the CPT margin legend.

Clock Selection
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the CS margin legend in the SYNOPSIS section.
Where additional semantics apply to a function, the material is identified by use of the CS margin legend.

Extension to the ISO C standard
The functionality described is an extension to the ISO C standard. Application writers may make use of an extension as it is supported on all IEEE Std 1003.1-2001-conforming systems.
With each function or header from the ISO C standard, a statement to the effect that “any conflict is unintentional” is included. That is intended to refer to a direct conflict. IEEE Std 1003.1-2001 acts in part as a profile of the ISO C standard, and it may choose to further constrain behaviors allowed to vary by the ISO C standard. Such limitations are not considered conflicts.
Where additional semantics apply to a function or header, the material is identified by use of the CX margin legend.

FORTRAN Development Utilities
The functionality described is optional.
Where applicable, utilities are marked with the FD margin legend in the SYNOPSIS section.
Where additional semantics apply to a utility, the material is identified by use of the FD margin legend.

FORTRAN Runtime Utilities
The functionality described is optional.
Where applicable, utilities are marked with the FR margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the FR margin legend.

**FSC**

*File Synchronization*

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

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

**IP6**

*IPV6*

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

Where applicable, functions are marked with the IP6 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the IP6 margin legend.

**MC1**

*Advisory Information and either Memory Mapped Files or Shared Memory Objects*

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

This is a shorthand notation for combinations of multiple option codes.

Where applicable, functions are marked with the MC1 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MC1 margin legend.

Refer to the Base Definitions volume of IEEE Std 1003.1-2001, Section 1.5.2, Margin Code Notation.

**MC2**

*Memory Mapped Files, Shared Memory Objects, or Memory Protection*

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

This is a shorthand notation for combinations of multiple option codes.

Where applicable, functions are marked with the MC2 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MC2 margin legend.

Refer to the Base Definitions volume of IEEE Std 1003.1-2001, Section 1.5.2, Margin Code Notation.

**MF**

*Memory Mapped Files*

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

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

**ML**

*Process Memory Locking*

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

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

194 margin legend.

195 **MLR**

196 **Range Memory Locking**

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

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

202 **MON**

203 **Monotonic Clock**

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

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

209 **MPR**

210 **Memory Protection**

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

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

216 **MSG**

217 **Message Passing**

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

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

223 **MX**

224 **IEC 60559 Floating-Point Option**

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

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

230 **OB**

231 **Obsolescent**

232 The functionality described may be withdrawn in a future version of this volume of
234 Applications shall not use obsolescent features.

235 Where applicable, the material is identified by use of the OB margin legend.

236 **OF**

237 **Output Format Incompletely Specified**

238 The functionality described is an XSI extension. The format of the output produced by the utility
239 is not fully specified. It is therefore not possible to post-process this output in a consistent
240 fashion. Typical problems include unknown length of strings and unspecified field delimiters.

241 Where applicable, the material is identified by use of the OF margin legend.

242 **OH**

243 **Optional Header**

244 In the SYNOPSIS section of some interfaces in the System Interfaces volume of
245 IEEE Std 1003.1-2001 an included header is marked as in the following example:
#include <sys/types.h>
#include <grp.h>

struct group *getgrnam(const char *name);

The OH margin legend indicates that the marked header is not required on XSI-conformant systems.

Prioritized Input and Output
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the PIO margin legend in the SYNOPSIS section.
Where additional semantics apply to a function, the material is identified by use of the PIO margin legend.

Process Scheduling
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the PS margin legend in the SYNOPSIS section.
Where additional semantics apply to a function, the material is identified by use of the PS margin legend.

Raw Sockets
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the RS margin legend in the SYNOPSIS section.
Where additional semantics apply to a function, the material is identified by use of the RS margin legend.

Realtime Signals Extension
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the RTS margin legend in the SYNOPSIS section.
Where additional semantics apply to a function, the material is identified by use of the RTS margin legend.

Software Development Utilities
The functionality described is optional.
Where applicable, utilities are marked with the SD margin legend in the SYNOPSIS section.
Where additional semantics apply to a utility, the material is identified by use of the SD margin legend.

Semaphores
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the SEM margin legend in the SYNOPSIS section.
Where additional semantics apply to a function, the material is identified by use of the SEM margin legend.

Shared Memory Objects
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the SHM margin legend in the SYNOPSIS section.
Where additional semantics apply to a function, the material is identified by use of the SHM
Portability Introduction

margin legend.

SIO
Synchronized Input and Output
The functionality described is optional. The functionality described is also an extension to the
ISO C standard.

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

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

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

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

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

SS
Process Sporadic Server
The functionality described is optional. The functionality described is also an extension to the
ISO C standard.

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

TCT
Thread CPU-Time Clocks
The functionality described is optional. The functionality described is also an extension to the
ISO C standard.

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

TEF
Trace Event Filter
The functionality described is optional. The functionality described is also an extension to the
ISO C standard.

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

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

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

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

8 System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
ISO C standard.

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

**TMR**
Timers
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the TMR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TMR margin legend.

**TPI**
Thread Priority Inheritance
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the TPI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPI margin legend.

**TPP**
Thread Priority Protection
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the TPP margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPP margin legend.

**TPS**
Thread Execution Scheduling
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the TPS margin legend for the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPS margin legend.

**TRC**
Trace
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the TRC margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRC margin legend.

**TRI**
Trace Inherit
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the TRI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRI margin legend.

**TRL**
Trace Log
The functionality described is optional. The functionality described is also an extension to the ISO C standard.
Where applicable, functions are marked with the TRL margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRL
margin legend.

TSA  
Thread Stack Address Attribute
The functionality described is optional. The functionality described is also an extension to the
ISO C standard.

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

TSF  
Thread-Safe Functions
The functionality described is optional. The functionality described is also an extension to the
ISO C standard.

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

TSH  
Thread Process-Shared Synchronization
The functionality described is optional. The functionality described is also an extension to the
ISO C standard.

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

TSP  
Thread Sporadic Server
The functionality described is optional. The functionality described is also an extension to the
ISO C standard.

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

TSS  
Thread Stack Address Size
The functionality described is optional. The functionality described is also an extension to the
ISO C standard.

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

TYM  
Typed Memory Objects
The functionality described is optional. The functionality described is also an extension to the
ISO C standard.

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

UP  
User Portability Utilities
The functionality described is optional.

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

XSI  
Extension
The functionality described is an XSI extension. Functionality marked XSI is also an extension to
the ISO C standard. Application writers may confidently make use of an extension on all
systems supporting the X/Open System Interfaces Extension.

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

**XSI STREAMS**

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

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

### 1.9 Format of Entries

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

**NAME**

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

**SYNOPSIS**

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

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

**DESCRIPTION**

This section describes the functionality of the function or header.

**RETURN VALUE**

This section indicates the possible return values, if any.

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

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

**ERRORS**

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

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

**EXAMPLES**

This section is informative.

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

This section is informative.

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

RATIONALE

This section is informative.

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

FUTURE DIRECTIONS

This section is informative.

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

SEE ALSO

This section is informative.

This section gives references to related information.

CHANGE HISTORY

This section is informative.

This section shows the derivation of the entry and any significant changes that have been made to it.
Chapter 2

General Information

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

2.1 Use and Implementation of Functions

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

1. If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behavior is undefined.

2. Any function declared in a header may also be implemented as a macro defined in the header, so a function should not be declared explicitly if its header is included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a function even if it is also defined as a macro. The use of the C-language #undef construct to remove any such macro definition shall also ensure that an actual function is referred to.

3. Any invocation of a function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments. Likewise, those function-like macros described in the following sections may be invoked in an expression anywhere a function with a compatible return type could be called.

4. Provided that a function can be declared without reference to any type defined in a header, it is also permissible to declare the function explicitly and use it without including its associated header.

5. If a function that accepts a variable number of arguments is not declared (explicitly or by including its associated header), the behavior is undefined.

2.2 The Compilation Environment

2.2.1 POSIX.1 Symbols

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

Symbols called “feature test macros” are used to control the visibility of symbols that might be included in a header. Implementations, future versions of IEEE Std 1003.1-2001, and other standards may define additional feature test macros.
In the compilation of an application that `#defines` a feature test macro specified by IEEE Std 1003.1-2001, no header defined by IEEE Std 1003.1-2001 shall be included prior to the definition of the feature test macro. This restriction also applies to any implementation-provided header in which these feature test macros are used. If the definition of the macro does not precede the `#include`, the result is undefined.

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

### 2.2.1.1 The `_POSIX_C_SOURCE` Feature Test Macro

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

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

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

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

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

### 2.2.1.2 The `_XOPEN_SOURCE` Feature Test Macro

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

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

### 2.2.2 The Name Space

All identifiers in this volume of IEEE Std 1003.1-2001, except `environ`, are defined in at least one of the headers, as shown in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers. When `_XOPEN_SOURCE` or `_POSIX_C_SOURCE` is defined, each header defines or declares some identifiers, potentially conflicting with identifiers used by the application. The set of identifiers visible to the application consists of precisely those identifiers from the header pages of the included headers, as well as additional identifiers reserved for the implementation. In addition, some headers may make visible identifiers from other headers as indicated on the relevant header pages.
Implementations may also add members to a structure or union without controlling the visibility of those members with a feature test macro, as long as a user-defined macro with the same name cannot interfere with the correct interpretation of the program. The identifiers reserved for use by the implementation are described below:

1. Each identifier with external linkage described in the header section is reserved for use as an identifier with external linkage if the header is included.

2. Each macro described in the header section is reserved for any use if the header is included.

3. Each identifier with file scope described in the header section is reserved for use as an identifier with file scope in the same name space if the header is included.

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

<table>
<thead>
<tr>
<th>Header</th>
<th>Prefix</th>
<th>Suffix</th>
<th>Complete Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;aio.h&gt;</td>
<td>aio_, lio_, AIO_, LIO_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;arpa/inet.h&gt;</td>
<td>in_, inet_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;ctype.h&gt;</td>
<td>to[a-z], is[a-z]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;dirent.h&gt;</td>
<td>d_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;errno.h&gt;</td>
<td>E[0-9], E[A-Z]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;fcntl.h&gt;</td>
<td><em>l</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;glob.h&gt;</td>
<td>gl_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;grp.h&gt;</td>
<td>gr_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;limits.h&gt;</td>
<td>_MAX, _MIN</td>
<td></td>
<td>int[0-9a-z]_<em><em>t, uint[0-9a-z]</em></em>_t</td>
</tr>
<tr>
<td>&lt;locale.h&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;loc.h&gt;</td>
<td>LC_[A-Z]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;mqueue.h&gt;</td>
<td>mq_, MQ_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;ndbm.h&gt;</td>
<td>dbm_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;netdb.h&gt;</td>
<td>h_, n_, p_, s_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;net/if.h&gt;</td>
<td>if_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;netinet/in.h&gt;</td>
<td>in_, ip_, s_, sin_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;poll.h&gt;</td>
<td>in6_, s6_, sin6_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;pthread.h&gt;</td>
<td>pthread_, PTHREAD_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;pwd.h&gt;</td>
<td>pw_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;regex.h&gt;</td>
<td>re_, rm_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;sched.h&gt;</td>
<td>sched_, SCHED_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;semaphore.h&gt;</td>
<td>sem_, SEM_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;signal.h&gt;</td>
<td>sa_, uc_, SIG[A-Z], SIG_[A-Z]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;stropts.h&gt;</td>
<td>si_, sl_, sigev_, SIGEV_, sival_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;stdlib.h&gt;</td>
<td>bi_, ic_, l_, sl_, str_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;stdio.h&gt;</td>
<td>str[a-z]</td>
<td></td>
<td>int[0-9a-z]_<em><em>t, uint[0-9a-z]</em></em>_t</td>
</tr>
<tr>
<td>&lt;string.h&gt;</td>
<td>str[a-z], mem[a-z], wcs[a-z]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;sys/ipc.h&gt;</td>
<td>ipc_</td>
<td></td>
<td>key, pad, seq</td>
</tr>
<tr>
<td>&lt;sys/mman.h&gt;</td>
<td>shm_, MAP_, MCL_, MS_, PROT_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;sys/msg.h&gt;</td>
<td>msg</td>
<td></td>
<td>msg</td>
</tr>
<tr>
<td>&lt;sys/resource.h&gt;</td>
<td>rlim_, ru_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;sys/select.h&gt;</td>
<td>fd_, fds_, FD_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;sys/sem.h&gt;</td>
<td>sem</td>
<td></td>
<td>mem</td>
</tr>
<tr>
<td>&lt;sys/shm.h&gt;</td>
<td>shm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;sys/socket.h&gt;</td>
<td>ss_, sa_, ifc_, ifru_, infu_, ifra_, msg_, cmmsg_, l_, st_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;sys/stat.h&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;sys/statvfs.h&gt;</td>
<td>f_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;sys/time.h&gt;</td>
<td>fds_, it_, tv_, FD_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;sys/times.h&gt;</td>
<td>tms_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header</td>
<td>Prefix</td>
<td>Suffix</td>
<td>Complete Name</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------</td>
<td>--------</td>
<td>------------------------</td>
</tr>
<tr>
<td><code>&lt;sys/uio.h&gt;</code></td>
<td>iov_</td>
<td></td>
<td>UIO_MAXIOV</td>
</tr>
<tr>
<td><code>&lt;sys/un.h&gt;</code></td>
<td>sun_</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;sys/utsname.h&gt;</code></td>
<td>uts_</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;sys/wait.h&gt;</code></td>
<td>si_, W[A-Z], P_</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;termios.h&gt;</code></td>
<td>c_</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;time.h&gt;</code></td>
<td>tm_</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;ucontext.h&gt;</code></td>
<td>clock_, timer_, it_, tv_,</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;ulimit.h&gt;</code></td>
<td>uc_, ss_</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;utime.h&gt;</code></td>
<td>utim_</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;utmpx.h&gt;</code></td>
<td>ut_</td>
<td>_LVL, _TIME, _PROCESS</td>
<td></td>
</tr>
<tr>
<td><code>&lt;wchar.h&gt;</code></td>
<td>wcs[a-z]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;wctype.h&gt;</code></td>
<td>is[a-z], to[a-z]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;wordexp.h&gt;</code></td>
<td>we_</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANY header</td>
<td>POSIX_, <em>POSIX</em>, posix_,</td>
<td>_t</td>
<td></td>
</tr>
</tbody>
</table>

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

If any header in the following table is included, macros with the prefixes shown may be defined. After the last inclusion of a given header, an application may use identifiers with the corresponding prefixes for its own purpose, provided their use is preceded by a `#undef` of the corresponding macro.
<table>
<thead>
<tr>
<th>Header</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;dlfcn.h&gt;</code></td>
<td>RTLD_</td>
</tr>
<tr>
<td><code>&lt;cntl.h&gt;</code></td>
<td>F_, O_, S_</td>
</tr>
<tr>
<td><code>&lt;fmtmsg.h&gt;</code></td>
<td>MM_</td>
</tr>
<tr>
<td><code>&lt;fnmatch.h&gt;</code></td>
<td>FNM_</td>
</tr>
<tr>
<td><code>&lt;ftw.h&gt;</code></td>
<td>FTW</td>
</tr>
<tr>
<td><code>&lt;glob.h&gt;</code></td>
<td>GLOB_</td>
</tr>
<tr>
<td><code>&lt;inttypes.h&gt;</code></td>
<td>PRI[a-z], SCN[a-z]</td>
</tr>
<tr>
<td><code>&lt;ndbm.h&gt;</code></td>
<td>DBM_</td>
</tr>
<tr>
<td><code>&lt;netinet/in.h&gt;</code></td>
<td>IMPLINK_, IN_, INADDR_, IP_, IPPORT_, IPPROTO_, SOCK_</td>
</tr>
<tr>
<td><code>&lt;netinet/tcp.h&gt;</code></td>
<td>TCP_</td>
</tr>
<tr>
<td><code>&lt;poll.h&gt;</code></td>
<td>POLL</td>
</tr>
<tr>
<td><code>&lt;regex.h&gt;</code></td>
<td>REG_</td>
</tr>
<tr>
<td><code>&lt;signal.h&gt;</code></td>
<td>SA_, SIG_[0-9a-z_]</td>
</tr>
<tr>
<td><code>&lt;stropts.h&gt;</code></td>
<td>FLUSH[A-Z], I_, M_, MUXID_R[A-Z], S_, SND[A-Z], STR</td>
</tr>
<tr>
<td><code>&lt;syslog.h&gt;</code></td>
<td>LOG_</td>
</tr>
<tr>
<td><code>&lt;sys/ipc.h&gt;</code></td>
<td>IPC_</td>
</tr>
<tr>
<td><code>&lt;sys/msg.h&gt;</code></td>
<td>MSG[A-Z]</td>
</tr>
<tr>
<td><code>&lt;sys/resource.h&gt;</code></td>
<td>PRIO_, RLIM_, RLIMIT_, RUSAGE_</td>
</tr>
<tr>
<td><code>&lt;sys/wait.h&gt;</code></td>
<td>BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SL_, TRAP_</td>
</tr>
<tr>
<td><code>&lt;sys/socket.h&gt;</code></td>
<td>AF_, CMSG_, MSG_, PF_, SCM_, SHUT_, SO</td>
</tr>
<tr>
<td><code>&lt;sys/stat.h&gt;</code></td>
<td>S_</td>
</tr>
<tr>
<td><code>&lt;sys/statvfs.h&gt;</code></td>
<td>ST_</td>
</tr>
<tr>
<td><code>&lt;sys/time.h&gt;</code></td>
<td>FD_, ITIMER_</td>
</tr>
<tr>
<td><code>&lt;sys/uio.h&gt;</code></td>
<td>IOV_</td>
</tr>
<tr>
<td><code>&lt;termios.h&gt;</code></td>
<td>V, I, O, TC, B[0-9] (See below.)</td>
</tr>
<tr>
<td><code>&lt;wordexp.h&gt;</code></td>
<td>WRDE_</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Reserved Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBAUD</td>
<td>EXTB</td>
</tr>
<tr>
<td>DEFECO</td>
<td>FLUSHO</td>
</tr>
<tr>
<td>ECHOCCTL</td>
<td>LOBLK</td>
</tr>
<tr>
<td>ECHOKE</td>
<td>PENDDIN</td>
</tr>
<tr>
<td>ECHOPRT</td>
<td>SWTC</td>
</tr>
<tr>
<td>EXTA</td>
<td>VDISCARD</td>
</tr>
</tbody>
</table>

18 System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
The following identifiers are reserved regardless of the inclusion of headers:

1. All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use by the implementation.

2. All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary identifier and tag name spaces.

3. All identifiers in the table below are reserved for use as identifiers with external linkage. Some of these identifiers do not appear in this volume of IEEE Std 1003.1-2001, but are reserved for future use by the ISO C standard.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
<th>Alias</th>
<th>Alias</th>
<th>Alias</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Exit</td>
<td></td>
<td>fabsf</td>
<td>lgammal</td>
<td>scalbln</td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td></td>
<td>fabsl</td>
<td>llabs</td>
<td>scalblnf</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td></td>
<td>fclose</td>
<td>llrint</td>
<td>scalblnl</td>
<td></td>
</tr>
<tr>
<td>acos</td>
<td></td>
<td>fdim</td>
<td>llrintf</td>
<td>scalbn</td>
<td></td>
</tr>
<tr>
<td>acosf</td>
<td></td>
<td>fdimf</td>
<td>llrintl</td>
<td>scalbnf</td>
<td></td>
</tr>
<tr>
<td>acosh</td>
<td></td>
<td>fdiml</td>
<td>llround</td>
<td>scalbln</td>
<td></td>
</tr>
<tr>
<td>acoshf</td>
<td></td>
<td>feclexcept</td>
<td>llroundf</td>
<td>scanf</td>
<td></td>
</tr>
<tr>
<td>acoshl</td>
<td></td>
<td>fegetenv</td>
<td>llroundl</td>
<td>setbuf</td>
<td></td>
</tr>
<tr>
<td>acosl</td>
<td></td>
<td>fegetexceptflag</td>
<td>localeconv</td>
<td>setjmp</td>
<td></td>
</tr>
<tr>
<td>acosl</td>
<td></td>
<td>fegetround</td>
<td>localtime</td>
<td>setlocale</td>
<td></td>
</tr>
<tr>
<td>asctime</td>
<td></td>
<td>feholdexcept</td>
<td>log</td>
<td>setvbuf</td>
<td></td>
</tr>
<tr>
<td>asin</td>
<td></td>
<td>feof</td>
<td>log10</td>
<td>signal</td>
<td></td>
</tr>
<tr>
<td>asinf</td>
<td></td>
<td>feraisexcept</td>
<td>log10f</td>
<td>sin</td>
<td></td>
</tr>
<tr>
<td>asinh</td>
<td></td>
<td>ferror</td>
<td>log10l</td>
<td>sinf</td>
<td></td>
</tr>
<tr>
<td>asinhf</td>
<td></td>
<td>fesetenv</td>
<td>log1p</td>
<td>sinh</td>
<td></td>
</tr>
<tr>
<td>asinhl</td>
<td></td>
<td>fesetexceptflag</td>
<td>log1pf</td>
<td>sinhf</td>
<td></td>
</tr>
<tr>
<td>asinl</td>
<td></td>
<td>fesetround</td>
<td>log1pl</td>
<td>sinhl</td>
<td></td>
</tr>
<tr>
<td>asinl</td>
<td></td>
<td>fetestexcept</td>
<td>log2</td>
<td>sinl</td>
<td></td>
</tr>
<tr>
<td>atan</td>
<td></td>
<td>feupdateenv</td>
<td>log2f</td>
<td>sprintf</td>
<td></td>
</tr>
<tr>
<td>atan2</td>
<td></td>
<td>ffflush</td>
<td>log2l</td>
<td>sqrt</td>
<td></td>
</tr>
<tr>
<td>atan2f</td>
<td></td>
<td>fgetc</td>
<td>logb</td>
<td>sqrtf</td>
<td></td>
</tr>
<tr>
<td>atan2l</td>
<td></td>
<td>fgetpos</td>
<td>logbf</td>
<td>sqrtl</td>
<td></td>
</tr>
<tr>
<td>atanf</td>
<td></td>
<td>fgets</td>
<td>logbl</td>
<td>srand</td>
<td></td>
</tr>
<tr>
<td>atanf</td>
<td></td>
<td>fgetwc</td>
<td>logf</td>
<td>sscanf</td>
<td></td>
</tr>
<tr>
<td>atanh</td>
<td></td>
<td>fgetws</td>
<td>logl</td>
<td>str[a-zA-Z]*</td>
<td></td>
</tr>
<tr>
<td>atanh</td>
<td></td>
<td>floor</td>
<td>longjmp</td>
<td>strtof</td>
<td></td>
</tr>
<tr>
<td>atanhf</td>
<td></td>
<td>floorf</td>
<td>lrint</td>
<td>strtoimax</td>
<td></td>
</tr>
<tr>
<td>atanh</td>
<td></td>
<td>floorl</td>
<td>lrintf</td>
<td>strtold</td>
<td></td>
</tr>
<tr>
<td>atanl</td>
<td></td>
<td>fma</td>
<td>lrintl</td>
<td>strtoll</td>
<td></td>
</tr>
<tr>
<td>atanl</td>
<td></td>
<td>lround</td>
<td>lrtoull</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atexit</td>
<td></td>
<td>lroundf</td>
<td>strtoymax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atof</td>
<td></td>
<td>lroundl</td>
<td>strtoprintf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atoi</td>
<td></td>
<td>fmax</td>
<td>malloc</td>
<td>swscanf</td>
<td></td>
</tr>
<tr>
<td>atol</td>
<td></td>
<td>fmaxl</td>
<td>mblen</td>
<td>system</td>
<td></td>
</tr>
<tr>
<td>atoll</td>
<td></td>
<td>fmin</td>
<td>mbrelen</td>
<td>tan</td>
<td></td>
</tr>
<tr>
<td>bsearch</td>
<td></td>
<td>fminf</td>
<td>mbtowc</td>
<td>tanf</td>
<td></td>
</tr>
<tr>
<td>cabs</td>
<td></td>
<td>fminl</td>
<td>msbinit</td>
<td>tanh</td>
<td></td>
</tr>
<tr>
<td>cabsf</td>
<td></td>
<td>fmod</td>
<td>msbtowcs</td>
<td>tanhf</td>
<td></td>
</tr>
<tr>
<td>cabsl</td>
<td></td>
<td>fmodf</td>
<td>mbstowcs</td>
<td>tanhl</td>
<td></td>
</tr>
<tr>
<td>cacos</td>
<td></td>
<td>fmodl</td>
<td>mbtowc</td>
<td>tanl</td>
<td></td>
</tr>
</tbody>
</table>

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved. 19
4. All functions and external identifiers defined in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers are reserved for use as identifiers with external linkage.

Note: The notation [a–z] indicates any lowercase letter in the portable character set. The notation ’*’ indicates any combination of digits, letters in the portable character set, or underscore.
5. All the identifiers defined in this volume of IEEE Std 1003.1-2001 that have external linkage are always reserved for use as identifiers with external linkage.

No other identifiers are reserved.

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

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

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

### 2.3 Error Numbers

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

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

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

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

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

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

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

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

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

[E2BIG]
Argument list too long. The sum of 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.

or:
Lack of space in an output buffer.

or:
Argument is greater than the system-imposed maximum.

[EACCES]
Permission denied. An attempt was made to access a file in a way forbidden by its file access permissions.

[EADDRINUSE]
Address in use. The specified address is in use.

[EADDRNOTAVAIL]
Address not available. The specified address is not available from the local system.

[EAFNOSUPPORT]
Address family not supported. The implementation does not support the specified address family, or the specified address is not a valid address for the address family of the specified socket.

[EAGAIN]
Resource temporarily unavailable. This is a temporary condition and later calls to the same routine may complete normally.

[EALREADY]
Connection already in progress. A connection request is already in progress for the specified socket.

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

[EBADMSG]
Bad message. During a read(), getmsg(), getpmsg(), or ioctl() I_RECVFD request to a STREAMS device, a message arrived at the head of the STREAM that is inappropriate for the function receiving the message.

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

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

Error Numbers

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

or:

Bad Message. The implementation has detected a corrupted message.

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

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

ECHILD
No child process. A wait() or waitpid() function was executed by a process that had no existing or unwaited-for child process.

[ECOMMABORTED]
Connection aborted. The connection has been aborted.

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

[ECOMMRESET]
Connection reset. The connection was forcibly closed by the peer.

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

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

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

[EDQUOT]
Reserved.

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

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

[EFBIG]
File too large. The size of a file would exceed the maximum file size of an implementation or offset maximum established in the corresponding file description.
[EHOSTUNREACH]
Host is unreachable. The destination host cannot be reached (probably because the host is
down or a remote router cannot reach it).

[EIDRM]
Identifier removed. Returned during XSI interprocess communication if an identifier has
been removed from the system.

[EILSEQ]
Illegal byte sequence. A wide-character code has been detected that does not correspond to
a valid character, or a byte sequence does not form a valid wide-character code (defined in
the ISO C standard).

[EINPROGRESS]
Operation in progress. This code is used to indicate that an asynchronous operation has not
yet completed.

or:

O_NONBLOCK is set for the socket file descriptor and the connection cannot be
immediately established.

[EINTR]
Interrupted function call. An asynchronous signal was caught by the process during the
execution of an interruptible function. If the signal handler performs a normal return, the
interrupted function call may return this condition (see the Base Definitions volume of

[EINVAL]
Invalid argument. Some invalid argument was supplied; for example, specifying an
undefined signal in a signal() function or a kill() function.

[EIO]
Input/output error. Some physical input or output error has occurred. This error may be
reported on a subsequent operation on the same file descriptor. Any other error-causing
operation on the same file descriptor may cause the [EIO] error indication to be lost.

[EISCONN]
Socket is connected. The specified socket is already connected.

[EISDIR]
Is a directory. An attempt was made to open a directory with write mode specified.

[ELOOP]
Symbolic link loop. A loop exists in symbolic links encountered during pathname
resolution. This error may also be returned if more than {SYMLOOP_MAX} symbolic links
are encountered during pathname resolution.

[EMFILE]
Too many open files. An attempt was made to open more than the maximum number of file
descriptors allowed in this process.

[EMLINK]
Too many links. An attempt was made to have the link count of a single file exceed
{LINK_MAX}.

[EMSGSIZE]
Message too large. A message sent on a transport provider was larger than an internal
message buffer or some other network limit.
or:

Inappropriate message buffer length.

[EMULTIHOP]
Reserved.

[ENAMETOOLONG]
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}.

[ENETDOWN]
Network is down. The local network interface used to reach the destination is down.

[ENETRESET]
The connection was aborted by the network.

[ENETUNREACH]
Network unreachable. No route to the network is present.

[ENFILE]
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.

[ENOBUFS]
No buffer space available. Insufficient buffer resources were available in the system to perform the socket operation.

[ENODATA]
No message available. No message is available on the STREAM head read queue.

[ENODEV]
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.

[ENOENT]
No such file or directory. A component of a specified pathname does not exist, or the pathname is an empty string.

[ENOEXEC]
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.

[ENOLCK]
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.

[ENOLINK]
Reserved.

[ENOMEM]
Not enough space. The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

[ENOMEM]
No message of the desired type. The message queue does not contain a message of the
required type during XSI interprocess communication.

[ENOPROTOOPT]
Protocol not available. The protocol option specified to _socketopt() is not supported by the implementation.

[ENOSPC]
No space left on a device. During the _write() function on a regular file or when extending a directory, there is no free space left on the device.

[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.

[ENOSTR]
Not a STREAM. A STREAM function was attempted on a file descriptor that was not associated with a STREAMS device.

[ENOSYS]
Function not implemented. An attempt was made to use a function that is not available in this implementation.

[ENOTCONN]
Socket not connected. The socket is not connected.

[ENOTDIR]
Not a directory. A component of the specified pathname exists, but it is not a directory, when a directory was expected.

[ENOTEMPTY]
Directory not empty. A directory other than an empty directory was supplied when an empty directory was expected.

[ENOTSOCK]
Not a socket. The file descriptor does not refer to a socket.

[ENOTSUP]
Not supported. The implementation does not support this feature of the Realtime Option Group.

[ENOTTY]
Inappropriate I/O control operation. A control function has been attempted for a file or special file for which the operation is inappropriate.

[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.

[EOPNOTSUPP]
Operation not supported on socket. The type of socket (address family or protocol) does not support the requested operation.

[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.
[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.

[EPipe]
Broken pipe. A write was attempted on a socket, pipe, or FIFO for which there is no process
to read the data.

[EPROTO]
Protocol error. Some protocol error occurred. This error is device-specific, but is generally
not related to a hardware failure.

[EPROTONOSUPPORT]
Protocol not supported. The protocol is not supported by the address family, or the protocol
is not supported by the implementation.

[EPROTOTYPE]
Protocol wrong type for socket. The socket type is not supported by the protocol.

[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).

[ERofs]
Read-only file system. An attempt was made to modify a file or directory on a file system
that is read-only.

[ESPIPE]
Invalid seek. An attempt was made to access the file offset associated with a pipe or FIFO.

[ESRCh]
No such process. No process can be found corresponding to that specified by the given
process ID.

[ESTALE]
Reserved.

[ETIME]
STREAM ioctl() timeout. The timer set for a STREAMS ioctl() 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 ioctl() operation
is unspecified.

[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.

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

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

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

2.3.1 Additional Error Numbers
Additional implementation-defined error numbers may be defined in <errno.h>.

2.4 Signal Concepts

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

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

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

During the time between the generation of a signal and its delivery or acceptance, the signal is
said to be “pending”. Ordinarily, this interval cannot be detected by an application. However, a
signal can be “blocked” from delivery to a thread. If the action associated with a blocked signal
is anything other than to ignore the signal, and if that signal is generated for the thread, the
signal shall remain pending until it is unblocked, it is accepted when it is selected and returned
by a call to the sigwait() function, or the action associated with it is set to ignore the signal.
Signals generated for the process shall be delivered to exactly one of those threads within the
process which is in a call to a sigwait() function selecting that signal or has not blocked delivery
of the signal. If there are no threads in a call to a sigwait() function selecting that signal, and if all
threads within the process block delivery of the signal, the signal shall remain pending on the
process until a thread calls a sigwait() function selecting that signal, a thread unblocks delivery
of the signal, or the action associated with the signal is set to ignore the signal. If the action
associated with a blocked signal is to ignore the signal and if that signal is generated for the
process, it is unspecified whether the signal is discarded immediately upon generation or
remains pending.
Each thread has a “signal mask” that defines the set of signals currently blocked from delivery to it. The signal mask for a thread shall be initialized from that of its parent or creating thread, or from the corresponding thread in the parent process if the thread was created as the result of a call to fork(). The pthread_sigmask(), sigaction(), sigprocmask(), and sigsuspend() functions control the manipulation of the signal mask.

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

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

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

### 2.4.2 Realtime Signal Generation and Delivery

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

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

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>sigev_notify</td>
<td>Notification type.</td>
</tr>
<tr>
<td>int</td>
<td>sigev_signo</td>
<td>Signal number.</td>
</tr>
<tr>
<td>union signal</td>
<td>sigev_value</td>
<td>Signal value.</td>
</tr>
<tr>
<td>void(*) (unsigned)</td>
<td>sigev_notify_function</td>
<td>Notification function.</td>
</tr>
<tr>
<td>(pthread_attr_t*)</td>
<td>sigev_notify_attributes</td>
<td>Notification attributes.</td>
</tr>
</tbody>
</table>

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

- **SIGEV_NONE** No asynchronous notification shall be delivered when the event of interest occurs.
- **SIGEV_SIGNAL** The signal specified in sigev_signo shall be generated for the process when the event of interest occurs. If the implementation supports the Realtime Signals Extension option and if the SA_SIGINFO flag is set for that signal number, then the signal shall be queued to the process and the value
1193 specified in `sigev_value` shall be the `si_value` component of the generated
1194 signal. If SA_SIGINFO is not set for that signal number, it is unspecified
1195 whether the signal is queued and what value, if any, is sent.

1196 SIGEV_THREAD A notification function shall be called to perform notification.

1197 An implementation may define additional notification mechanisms.

1198 The `sigev_signo` member specifies the signal to be generated. The `sigev_value` member is the
1199 application-defined value to be passed to the signal-catching function at the time of the signal
delivery or to be returned at signal acceptance as the `si_value` member of the `siginfo_t` structure.

1200 The `signal` union is defined in `<signal.h>` and contains at least the following members:

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>sival_int</td>
<td>Integer signal value.</td>
</tr>
<tr>
<td>void*</td>
<td>sival_ptr</td>
<td>Pointer signal value.</td>
</tr>
</tbody>
</table>

1206 The `sival_int` member shall be used when the application-defined value is of type `int`; the
1207 `sival_ptr` member shall be used when the application-defined value is a pointer.

1208 When a signal is generated by the `sigqueue()` function or any signal-generating function that
1209 supports the specification of an application-defined value, the signal shall be marked pending
and, if the SA_SIGINFO flag is set for that signal, the signal shall be queued to the process along
1210 with the application-specified signal value. Multiple occurrences of signals so generated are
1211 queued in FIFO order. It is unspecified whether signals so generated are queued when the
1212 SA_SIGINFO flag is not set for that signal.

1213 Signals generated by the `kill()` function or other events that cause signals to occur, such as
detection of hardware faults, `alarm()` timer expiration, or terminal activity, and for which the
implementation does not support queuing, shall have no effect on signals already queued for the
same signal number.

1214 When multiple unblocked signals, all in the range SIGRTMIN to SIGRTMAX, are pending, the
behavior shall be as if the implementation delivers the pending unblocked signal with the lowest
signal number within that range. No other ordering of signal delivery is specified.

1215 If, when a pending signal is delivered, there are additional signals queued to that signal number,
the signal shall remain pending. Otherwise, the pending indication shall be reset.

1216 Multi-threaded programs can use an alternate event notification mechanism. When a
notification is processed, and the `sigev_notify` member of the `sigevent` structure has the value
SIGEV_THREAD, the function `sigev_notify_function` is called with parameter `sigev_value`.

1217 The function shall be executed in an environment as if it were the `start_routine` for a newly
created thread with thread attributes specified by `sigev_notify_attributes`. If `sigev_notify_attributes`
is NULL, the behavior shall be as if the thread were created with the `detachstate` attribute set to
PTHREAD_CREATE_DETACHED. Supplying an attributes structure with a `detachstate` attribute
of PTHREAD_CREATE_JOINABLE results in undefined behavior. The signal mask of this
thread is implementation-defined.
2.4.3 Signal Actions

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

SIG_DFL  Signal-specific default action.

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

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

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

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

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

SIG_IGN  Ignore signal.

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

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

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

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

If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the calling processes shall not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it shall block until all of its children terminate, and wait(), waitid(), and waitpid() shall fail and set errno to [ECHILD].
If the Realtime Signals Extension option is supported, any queued values pending shall be discarded and the resources used to queue them shall be released and made available to queue other signals.

Catch signal.

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

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

```c
void func(int signo);
```

If the SA_SIGINFO flag for the signal is set, the signal-catch function shall be entered as a C-language function call as follows:

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

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

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td><code>si_signo</code></td>
<td>Signal number.</td>
</tr>
<tr>
<td>int</td>
<td><code>si_code</code></td>
<td>Cause of the signal.</td>
</tr>
<tr>
<td>union signal</td>
<td><code>si_value</code></td>
<td>Signal value.</td>
</tr>
</tbody>
</table>

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

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

- **SI_QUEUE** The signal was sent by the `sigqueue()` function.

- **SI_TIMER** The signal was generated by the expiration of a timer set by `timer_settime()`.

- **SI_ASYNCIO** The signal was generated by the completion of an asynchronous I/O request.

- **SI_MESGQ** The signal was generated by the arrival of a message on an empty message queue.

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

If the Realtime Signals Extension is supported, and `si_code` is one of `SI_QUEUE`, `SI_TIMER`, `SI_ASYNCIO`, or `SI_MESGQ`, then `si_value` shall contain the application-specified signal value. Otherwise, the contents of `si_value` are undefined.
The behavior of a process is undefined after it returns normally from a signal-catch function for a SIGBUS, SIGFPE, SIGILL, or SIGSEGV signal that was not generated by \texttt{kill()}, \texttt{sigqueue()}, or \texttt{raise()}.

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

If a process establishes a signal-catch 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.

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

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

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

### 2.4.4 Signal Effects on Other Functions

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

### 2.5 Standard I/O Streams

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

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

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

A file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the main() function returns to its original caller, or if the exit() function is called, all open files are closed (hence all output streams are flushed) before program termination. Other paths to program termination, such as calling abort(), need not close all files properly.
The address of the FILE object used to control a stream may be significant; a copy of a FILE object need not necessarily serve in place of the original.

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

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

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

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

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

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

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

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

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

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

- If it is a file descriptor, no action is required.
- If the only further action to be performed on any handle to this open file descriptor is to close
  it, no action need be taken.
- If it is a stream which is unbuffered, no action need be taken.
- If it is a stream which is line buffered, and the last byte written to the stream was a
  `<newline>` (that is, as if a:
  
  ```c
  putc('
')
  ```
  
  was the most recent operation on that stream), no action need be taken.
- If it is a stream which is open for writing or appending (but not also open for reading), the
  application shall either perform an `fflush()`, or the stream shall be closed.
- If the stream is open for reading and it is at the end of the file (`feof()` is true), no action need
  be taken.
- If the stream is open with a mode that allows reading and the underlying open file
  description refers to a device that is capable of seeking, the application shall either perform
  an `fflush()`, or the stream shall be closed.

Otherwise, the result is undefined.

For the second handle:

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

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

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

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

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

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

For conformance to the ISO/IEC 9899:1999 standard, the definition of a stream includes an "orientation". After a stream is associated with an external file, but before any operations are performed on it, the stream is without orientation. Once a wide-character input/output function has been applied to a stream without orientation, the stream shall become "wide-oriented". Similarly, once a byte input/output function has been applied to a stream without orientation, the stream shall become "byte-oriented". Only a call to the fopen() function or the fwrite() function can otherwise alter the orientation of a stream.

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

Byte input/output functions cannot be applied to a wide-oriented stream, and wide-character input/output functions cannot be applied to a byte-oriented stream. The remaining stream operations shall not affect and shall not be affected by a stream’s orientation, except for the 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.

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

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

Although wide-oriented streams are conceptually sequences of wide characters, the external file associated with a wide-oriented stream is a sequence of (possibly multi-byte) characters 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.

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

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

The wide-character output functions convert wide characters to (possibly multi-byte) characters and write them to the stream as if they were written by successive calls to the fputwc() function. Each conversion shall occur as if by a call to the wctomb() function, with the conversion state described by the stream’s own mbstate_t object, except the encoding rule associated with the stream is used instead of the encoding rule implied by the LC_CTYPE category of the current locale.
An “encoding error” shall occur if the character sequence presented to the underlying `mbstrwc()` function does not form a valid (generalized) character, or if the code value passed to the underlying ` wcrtomb()` function does not correspond to a valid (generalized) character. The wide-character input/output functions and the byte input/output functions store the value of the macro `[EILSEQ]` in `errno` if and only if an encoding error occurs.

### 2.6 STREAMS

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

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

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

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

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

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

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

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

Message Parts

A process may access STREAMS messages that contain a data part, control part, or both. The data part is that information which is transmitted over the communication medium and the control information is used by the local STREAMS modules. The other types of messages are used between modules and are not accessible to processes. Messages containing only a data part are accessible via `putmsg()`, `putpmsg()`, `getmsg()`, `getpmsg()`, `read()`, `readv()`, `write()`, or `writev()`.

Messages containing a control part with or without a data part are accessible via calls to `putmsg()`, `putpmsg()`, `getmsg()`, or `getpmsg()`.

2.6.1 Accessing STREAMS

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

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

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

2.7 XSI Interprocess Communication

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

The following message passing, semaphore, and shared memory services form an XSI interprocess communication facility. Certain aspects of their operation are common, and are defined as follows.
Another interprocess communication facility is provided by functions in the Realtime Option Group; see Section 2.8 (on page 41).

### 2.7.1 IPC General Description

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

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

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

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0400</td>
<td>Read by user</td>
</tr>
<tr>
<td>0200</td>
<td>Write by user</td>
</tr>
<tr>
<td>0040</td>
<td>Read by group</td>
</tr>
<tr>
<td>0020</td>
<td>Write by group</td>
</tr>
<tr>
<td>0004</td>
<td>Read by others</td>
</tr>
<tr>
<td>0002</td>
<td>Write by others</td>
</tr>
</tbody>
</table>

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

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

2.8 Realtime

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

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

- Semaphores
- Process Memory Locking
- Memory Mapped Files and Shared Memory Objects
- Priority Scheduling
- Realtime Signal Extension
- Timers
- Interprocess Communication
- Synchronized Input and Output
- Asynchronous Input and Output

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

2.8.1 Realtime Signals

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

See Section 2.4.2 (on page 29).

2.8.2 Asynchronous I/O

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

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

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>aio_fildes</td>
<td>File descriptor.</td>
</tr>
<tr>
<td>off_t</td>
<td>aio_offset</td>
<td>File offset.</td>
</tr>
<tr>
<td>volatile</td>
<td>aio_buf</td>
<td>Location of buffer.</td>
</tr>
<tr>
<td>size_t</td>
<td>aio_nbytes</td>
<td>Length of transfer.</td>
</tr>
<tr>
<td>int</td>
<td>aio_reaprio</td>
<td>Request priority offset.</td>
</tr>
<tr>
<td>struct sigevent</td>
<td>aio_sigevent</td>
<td>Signal number and value.</td>
</tr>
<tr>
<td>int</td>
<td>aio_lio_opcode</td>
<td>Operation to be performed.</td>
</tr>
</tbody>
</table>
The `aio_fildes` element is the file descriptor on which the asynchronous operation is performed.

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

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

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

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

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

The `aiocb` structure and the data buffers associated with the asynchronous I/O operation are being used by the system for asynchronous I/O while, and only while, the error status of the
asynchronous operation is equal to [EINPROGRESS]. Applications shall not modify the \texttt{aiocb} structure while the structure is being used by the system for asynchronous I/O.

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

### 2.8.3 Memory Management

#### 2.8.3.1 Memory Locking

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

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

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

#### 2.8.3.2 Memory Mapped Files

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

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

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

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

An \texttt{unlink()} of a file or \texttt{shm_unlink()} of a shared memory object, while causing the removal of the name, does not unmap any mappings established for the object. Once the name has been removed, the contents of the memory object are preserved as long as it is referenced. The memory object remains referenced as long as a process has the memory object open or has some area of the memory object mapped.
2.8.3.3 Memory Protection

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

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

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

2.8.3.4 Typed Memory Objects

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

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

2.8.4 Process Scheduling

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

Scheduling Policies

The scheduling semantics described in this volume of IEEE Std 1003.1-2001 are defined in terms of a conceptual model that contains a set of thread lists. No implementation structures are necessarily implied by the use of this conceptual model. It is assumed that no time elapses during operations described using this model, and therefore no simultaneous operations are possible. This model discusses only processor scheduling for runnable threads, but it should be noted that greatly enhanced predictability of realtime applications results if the sequencing of other resources takes processor scheduling policy into account.
There is, conceptually, one thread list for each priority. A runnable thread will be on the thread list for that thread’s priority. Multiple scheduling policies shall be provided. Each non-empty thread list is ordered, contains a head as one end of its order, and a tail as the other. The purpose of a scheduling policy is to define the allowable operations on this set of lists (for example, moving threads between and within lists).

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

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

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

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

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

- **SCHED_FIFO** — First in, first out (FIFO) scheduling policy.
- **SCHED_RR** — Round robin scheduling policy.
- **SCHED_SPORADIC** — Sporadic server scheduling policy.
- **SCHED_OTHER** — Another scheduling policy.

The values of these symbols shall be distinct.

**SCHED_FIFO**

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

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

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

1. When a running thread becomes a preempted thread, it becomes the head of the thread list for its priority.
2. When a blocked thread becomes a runnable thread, it becomes the tail of the thread list for its priority.
3. When a running thread calls the `sched_setscheduler()` function, the process specified in the function call is modified to the specified policy and the priority specified by the `param` argument.
4. When a running thread calls the `sched_setparam()` function, the priority of the process specified in the function call is modified to the priority specified by the `param` argument.
5. When a running thread calls the `pthread_setschedparam()` function, the thread specified in the function call is modified to the specified policy and the priority specified by the `param` argument.

6. When a running thread calls the `pthread_setschedprio()` function, the thread specified in the function call is modified to the priority specified by the `prio` argument.

7. If a thread whose policy or priority has been modified other than by `pthread_setschedprio()` is a running thread or is runnable, it then becomes the tail of the thread list for its new priority.

8. If a thread whose policy or priority has been modified by `pthread_setschedprio()` is a running thread or is runnable, the effect on its position in the thread list depends on the direction of the modification, as follows:
   a. If the priority is raised, the thread becomes the tail of the thread list.
   b. If the priority is unchanged, the thread does not change position in the thread list.
   c. If the priority is lowered, the thread becomes the head of the thread list.

9. When a running thread issues the `sched_yield()` function, the thread becomes the tail of the thread list for its priority.

10. At no other time is the position of a thread with this scheduling policy within the thread lists affected.

For this policy, valid priorities shall be within the range returned by the `sched_get_priority_max()` and `sched_get_priority_min()` functions when SCHED_FIFO is provided as the parameter.

Conforming implementations shall provide a priority range of at least 32 priorities for this policy.

**SCHED_RR**

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

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

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

For this policy, valid priorities shall be within the range returned by the `sched_get_priority_max()` and `sched_get_priority_min()` functions when SCHED_RR is provided as the parameter.

Conforming implementations shall provide a priority range of at least 32 priorities for this policy.
The functionality described in this section is dependent on support of the Process Sporadic Server or Thread Sporadic Server options (and the rest of this section is not further shaded for these options).

If \_POSIX\_SPORADIC\_SERVER or \_POSIX\_THREAD\_SPORADIC\_SERVER is defined, the implementation shall include a scheduling policy identified by the value SCHED\_SPORADIC.

The sporadic server policy is based primarily on two parameters: the replenishment period and the available execution capacity. The replenishment period is given by the \texttt{sched\_ss\_repl\_period} member of the \texttt{sched\_param} structure. The available execution capacity is initialized to the value given by the \texttt{sched\_ss\_init\_budget} member of the same parameter. The sporadic server policy is identical to the SCHED\_FIFO policy with some additional conditions that cause the thread’s assigned priority to be switched between the values specified by the \texttt{sched\_priority} and \texttt{sched\_ss\_low\_priority} members of the \texttt{sched\_param} structure.

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

1. When the thread at the head of the \texttt{sched\_priority} list becomes a running thread, its execution time shall be limited to at most its available execution capacity, plus the resolution of the execution time clock used for this scheduling policy. This resolution shall be implementation-defined.

2. Each time the thread is inserted at the tail of the list associated with \texttt{sched\_priority}—because as a blocked thread it became runnable with priority \texttt{sched\_priority} or because a replenishment operation was performed—the time at which this operation is done is posted as the \texttt{activation\_time}.

3. When the running thread with assigned priority equal to \texttt{sched\_priority} becomes a preempted thread, it becomes the head of the thread list for its priority, and the execution time consumed is subtracted from the available execution capacity. If the available execution capacity would become negative by this operation, it shall be set to zero.

4. When the running thread with assigned priority equal to \texttt{sched\_priority} becomes a blocked thread, the execution time consumed is subtracted from the available execution capacity, and a replenishment operation is scheduled, as described in 6 and 7. If the available execution capacity would become negative by this operation, it shall be set to zero.

5. When the running thread with assigned priority equal to \texttt{sched\_priority} reaches the limit imposed on its execution time, it becomes the tail of the thread list for \texttt{sched\_ss\_low\_priority}, the execution time consumed is subtracted from the available execution capacity (which becomes zero), and a replenishment operation is scheduled, as described in 6 and 7.

6. Each time a replenishment operation is scheduled, the amount of execution capacity to be replenished, \texttt{replenish\_amount}, is set equal to the execution time consumed by the thread since the \texttt{activation\_time}. The replenishment is scheduled to occur at \texttt{activation\_time} plus \texttt{sched\_ss\_repl\_period}. If the scheduled time obtained is before the current time, the
replenishment operation is carried out immediately. Several replenishment operations may
be pending at the same time, each of which will be serviced at its respective scheduled
time. With the above rules, the number of replenishment operations simultaneously
pending for a given thread that is scheduled under the sporadic server policy shall not be
greater than sched_ss_max_repl.

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

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

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

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

SCHED_OTHER

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

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

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

2.8.5 Clocks and Timers

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

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

Time Value Specification Structures

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

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time_t</td>
<td>tv_sec</td>
<td>Seconds.</td>
</tr>
<tr>
<td>long</td>
<td>tv_nsec</td>
<td>Nanoseconds.</td>
</tr>
</tbody>
</table>

The tv_nsec member is only valid if greater than or equal to zero, and less than the number of
nanoseconds in a second (1 000 million). The time interval described by this structure is (tv_sec *
$10^9 + tv_nsec$) nanoseconds.
A time value structure `itimerspec` specifies an initial timer value and a repetition interval for use by the per-process timer functions. This structure includes at least the following members:

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>struct timespec</code></td>
<td><code>it_interval</code></td>
<td>Timer period.</td>
</tr>
<tr>
<td><code>struct timespec</code></td>
<td><code>it_value</code></td>
<td>Timer expiration.</td>
</tr>
</tbody>
</table>

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

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

**Timer Event Notification Control Block**

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

**Manifest Constants**

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

- `CLOCK_REALTIME`: The identifier for the system-wide realtime clock.
- `TIMER_ABSTIME`: Flag indicating time is absolute with respect to the clock associated with a timer.
- `CLOCK_MONOTONIC`: The identifier for the system-wide monotonic clock, which is defined as a clock whose value cannot be set via `clock_settime()` and which cannot have backward clock jumps. The maximum possible clock jump is implementation-defined.
- `CLOCK_MONOTONIC`: The maximum allowable resolution for `CLOCK_REALTIME` and `CLOCK_MONOTONIC` clocks and all time services based on these clocks is represented by `_POSIX_CLOCKRES_MIN` and shall be defined as 20 ms (1/50 of a second). Implementations may support smaller values of resolution for these clocks to provide finer granularity time bases. The actual resolution supported by an implementation for a specific clock is obtained using the `clock_getres()` function. If the actual resolution supported for a time service based on one of these clocks differs from the resolution supported for that clock, the implementation shall document this difference.

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

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

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

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

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

```
CLOCK_PROCESS_CPUTIME_ID
```

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

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

```
CLOCK_THREAD_CPUTIME_ID
```

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

2.9 Threads

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

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

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

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

2.9.1 Thread-Safety

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

---

¹ The functions in the table are not shaded to denote applicable options. Individual reference pages should be consulted.
The `ctermid`() and `tmpnam`() functions need not be thread-safe if passed a NULL argument. The `wctomb`() and `wcsctomb`() functions need not be thread-safe if passed a NULL `ps` argument.

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

### 2.9.2 Thread IDs

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

### 2.9.3 Thread Mutexes

A thread that has blocked 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 is determined by the scheduling policy.

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

- It returns successfully from `pthread_mutex_lock()` with `m` as the `mutex` argument.
- It returns successfully from `pthread_mutex_trylock()` with `m` as the `mutex` argument.
- It returns successfully from `pthread_mutex_timedlock()` with `m` as the `mutex` argument.
- It returns (successfully or not) from `pthread_cond_wait()` with `m` as the `mutex` argument (except as explicitly indicated otherwise for certain errors).
- It returns (successfully or not) from `pthread_cond_timedwait()` with `m` as the `mutex` argument (except as explicitly indicated otherwise for certain errors).

The thread shall remain the owner of `m` until one of the following occurs:
• It executes `pthread_mutex_unlock()` with `m` as the mutex argument
• It blocks in a call to `pthread_cond_wait()` with `m` as the mutex argument.
• It blocks in a call to `pthread_cond_timedwait()` with `m` as the mutex argument.

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

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

### 2.9.4 Thread Scheduling

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

#### Thread Scheduling Attributes

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

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

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

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

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

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

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

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

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

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

- Threads created with PTHREAD_SCOPE_PROCESS scheduling contention scope contend directly with other threads within their process that were created with PTHREAD_SCOPE_PROCESS scheduling contention scope. The contention is resolved based on the threads’ scheduling attributes and policies. It is unspecified how such threads are scheduled relative to threads in other processes or threads with PTHREAD_SCOPE_SYSTEM scheduling contention scope.

- Conforming implementations shall support the PTHREAD_SCOPE_PROCESS scheduling contention scope, the PTHREAD_SCOPE_SYSTEM scheduling contention scope, or both.

Scheduling Allocation Domain

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

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

For application threads with scheduling allocation domains of size equal to one, the scheduling rules defined for SCHED_FIFO and SCHED_RR shall be used; see Scheduling Policies (on page 4). All threads with system scheduling contention scope, regardless of the processes in which they reside, compete for the processor according to their priorities. Threads with process scheduling contention scope compete only with other threads with process scheduling contention scope within their process.
For application threads with scheduling allocation domains of size greater than one, the rules
defined for SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC shall be used in an
implementation-defined manner. Each thread with system scheduling contention scope
competes for the processors in its scheduling allocation domain in an implementation-defined
manner according to its priority. Threads with process scheduling contention scope are
scheduled relative to other threads within the same scheduling contention scope in the process.

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

Scheduling Documentation

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

2.9.5 Thread Cancelation

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

Cancelation is controlled by the cancelation control functions. Each thread maintains its own
cancelability state. Cancelation may only occur at cancelation points or when the thread is
asynchronously cancelable.

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

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

2.9.5.1 Cancelability States

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

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

1. Cancelability-Enable: When cancelability is PTHREAD_CANCEL_DISABLE (as defined in
   the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>), cancelation requests
   against the target thread are held pending. By default, cancelability is set to
   PTHREAD_CANCEL_ENABLE (as defined in <pthread.h>).

2. Cancelability Type: When cancelability is enabled and the cancelability type is
   PTHREAD_CANCELASYNCHRONOUS (as defined in <pthread.h>), new or pending
   cancelation requests may be acted upon at any time. When cancelability is enabled and the
   cancelability type is PTHREAD_CANCELDDEFERRED (as defined in <pthread.h>),
cancelation requests are held pending until a cancelation point (see below) is reached. If cancelability is disabled, the setting of the cancelability type has no immediate effect as all cancelation requests are held pending; however, once cancelability is enabled again the new type is in effect. The cancelability type is PTHREAD_CANCEL_DEFERRED in all newly created threads including the thread in which main() was first invoked.

2.9.5.2 Cancelation Points

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

- `accept()`, `mq_timedsend()`, `putpmsg()`, `sigsuspend()`
- `aio_suspend()`, `msgrcv()`, `pwrite()`, `sigtimedwait()`
- `clock_nanosleep()`, `msgsnd()`, `read()`, `sigwait()`
- `close()`, `msync()`, `readv()`, `sigwaitinfo()`
- `connect()`, `nanosleep()`, `recv()`, `sleep()`
- `creat()`, `open()`, `recvfrom()`, `system()`
- `fcntl()`<sup>2</sup>, `pause()`, `recvmsg()`, `tcdrain()`
- `fsync()`, `poll()`, `select()`, `usleep()`
- `getmsg()`, `pread()`, `sem_timedwait()`, `wait()`
- `getpmsg()`, `pthread_cond_timedwait()`, `sem_wait()`, `waitid()`
- `lockf()`, `pthread_cond_wait()`, `send()`, `waitpid()`
- `mq_receive()`, `pthread_join()`, `sendmsg()`, `write()`
- `mq_send()`, `pthread_testcancel()`, `sendto()`, `writev()`
- `mq_timedreceive()`, `putmsg()`, `sigpause()`
A cancelation point may also occur when a thread is executing the following functions:

\begin{verbatim}
close()  ftell()  getwc()  pthread_rwlock_wrlock()
catgets()  ftello()  getwchar()  putc()
catopen()  ftw()  getwd()  putc_unlocked()
close()  fwrite()  fclose()
closelog()  frewrite()  iconv_close()  putchar_unlocked()
close()  fscanf()  iconv_open()  puts()
dbm_close()  getc()  ioctl()  pututsline()
dbm_delete()  getc_unlocked()  lseek()  putwc()
dbm_fetch()  getchar()  mkstemp()  putwchar()
dbm_nextkey()  getchar_unlocked()  nf�크()  readdir()
dbm_open()  getcwd()  opendir()  readdir_r()
dbm_store()  getdate()  openlog()  rename()
dlclose()  getgrent()  pclose()  rmdir()
dlopen()  getgrgid()  perror()  rewind()
dlerror()  getgrgid_r()  popen()  rewinddir()
dendhostent()  getgrnam()  posix_fadvise()  scanf()
ednetent()  getgrnam_r()  posix_fallocate()  seekdir()
edprotoent()  getgthbyaddr()  posix_fadvise()  semap()
endpwnent()  gethostbyname()  posix_spawn()  setgrent()
endservent()  gethostent()  posix_spawnp()  sethsetent()
endutxent()  gethostbyname()  posix_trace_clear()  setnetent()
fcntl()  getlogin()  posix_trace_close()  setprotoent()
fclose()  getlogin_r()  posix_trace_create()  setprocpt()
fchdir()  getnetbyaddr()  posix_trace_create_withlog()  setsetent()
ftime()  getnetbyaddr()  posix_trace_eventtypelist_getnext_id()  settutxent()
getctime()  getnetbyaddr()  posix_trace_eventtypelist_rewind()  sterror()
getdtablesize()  getprotobyname()  posix_trace_flush()  syslog()
getdomainname()  getprotobyname()  posix_trace_get_attr()  tmpfile()
getenv()  getprotobynumber()  posix_trace_get_filter()  tmpnam()
getfsize()  getprotot()  posix_trace_get_status()  ttyname()
getgsize()  getpwnam()  posix_trace_getnext_event()  ttyname_r()
getltext()  getpwnam_r()  posix_trace_open()  ungetc()
getopt()  getpwuid()  posix_trace_rewind()  ungetwc()
getpwnam()  getpwuid()  posix_trace_set_filter()  unlink()
getpwnam_r()  getpwuid()  posix_trace_setptr()  unlink()
getpwnam()  gets()  posix_trace_shutdown()  vfprintf()
getrsize()  getrsize()  posix_trace_timegetnext_event()  vfprintf()
getsockname()  getservbyname()  posix_type_mem_open()  vfprintf()
getsockname()  getservbyprot()  posix_type_mem_open()  vfprintf()
getsockopt()  getservent()  printf()  wfprintf()
gettimeofday()  getutxent()  pthraed_rwlock_rdlock()  wprintf()
getutixid()  getutxid()  pthread_rwlock_timedrdlock()  wscanf()
setutxid()  getutxid()  pthread_rwlock_timedwrlock()
\end{verbatim}

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

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

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

### 2.9.5.3 Thread Cancelation Cleanup Handlers

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

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

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

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

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

### 2.9.5.4 Async-Cancel Safety

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

No other functions in this volume of IEEE Std 1003.1-2001 are required to be async-cancel-safe.
2.9.6 Thread Read-Write Locks

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

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

A thread that has blocked on a read-write lock (for example, has not yet returned from a pthread_rwlock_rdlock() or pthread_rwlock_wrlock() call) 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.

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

2.9.7 Thread Interactions with Regular File Operations

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

2.10 Sockets

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

2.10.1 Address Families

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

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

An address family defines the format of a socket address. All network addresses are described using a general structure, called a `sockaddr`, as defined in the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/socket.h>`. However, each address family imposes finer and more specific structure, generally defining a structure with fields specific to the address family. The field `sa_family` in the `sockaddr` structure contains the address family identifier, specifying the 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
actually sent or received. However, implementations should attempt to detect any errors
possible before the return of an output function, reporting any error by an unsuccessful return
value.

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

### 2.10.7 Socket I/O Mode

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

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

### 2.10.8 Socket Owner

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

### 2.10.9 Socket Queue Limits

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

### 2.10.10 Pending Error

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

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

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

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

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

2.10.12 Socket Out-of-Band Data State

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

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

2.10.14 Signals

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

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

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

2.10.15 Asynchronous Errors

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

- **[ECONNABORTED]**
  - The connection was aborted locally.

- **[ECONNREFUSED]**
  - For a connection-mode socket attempting a non-blocking connection, the attempt to connect was forcefully rejected. For a connectionless-mode socket, an attempt to deliver a datagram was forcefully rejected.

- **[ECONNRESET]**
  - The peer has aborted the connection.

- **[EHOSTDOWN]**
  - The destination host has been determined to be down or disconnected.

- **[EHOSTUNREACH]**
  - The destination host is not reachable.

- **[EMSGSIZE]**
  - For a connectionless-mode socket, the size of a previously sent datagram prevented delivery.

- **[ENETDOWN]**
  - The local network connection is not operational.

- **[ENETRESET]**
  - The connection was aborted by the network.

- **[ENETUNREACH]**
  - The destination network is not reachable.
2.10.16 Use of Options

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

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

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

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOL_SOCKET</td>
<td>Options are intended for the sockets level.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Parameter Type</th>
<th>Parameter Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SO_BROADCAST</td>
<td>int</td>
<td>Non-zero requests permission to transmit broadcast datagrams (SOCK_DGRAM sockets only).</td>
</tr>
<tr>
<td>SO_DEBUG</td>
<td>int</td>
<td>Non-zero requests debugging in underlying protocol modules.</td>
</tr>
<tr>
<td>SO_DONTROUTE</td>
<td>int</td>
<td>Non-zero requests bypass of normal routing; route based on destination address only.</td>
</tr>
<tr>
<td>SO_ERROR</td>
<td>int</td>
<td>Requests and clears pending error information on the socket (getsockopt() only).</td>
</tr>
<tr>
<td>SO_KEEPALIVE</td>
<td>int</td>
<td>Non-zero requests periodic transmission of keepalive messages (protocol-specific).</td>
</tr>
<tr>
<td>SO_LINGER</td>
<td>struct linger</td>
<td>Specify actions to be taken for queued, unsent data on close(): linger on/off and linger time in seconds.</td>
</tr>
<tr>
<td>SO_OOBINLINE</td>
<td>int</td>
<td>Non-zero requests that out-of-band data be placed into normal data input queue as received.</td>
</tr>
<tr>
<td>SO_RCVBUF</td>
<td>int</td>
<td>Size of receive buffer (in bytes).</td>
</tr>
<tr>
<td>SO_RCVLOWAT</td>
<td>int</td>
<td>Minimum amount of data to return to application for input operations (in bytes).</td>
</tr>
<tr>
<td>SO_RCVTIMEO</td>
<td>struct timeval</td>
<td>Timeout value for a socket receive operation.</td>
</tr>
<tr>
<td>SO_REUSEADDR</td>
<td>int</td>
<td>Non-zero requests reuse of local addresses in bind() (protocol-specific).</td>
</tr>
<tr>
<td>SO_SNDBUF</td>
<td>int</td>
<td>Size of send buffer (in bytes).</td>
</tr>
<tr>
<td>SO_SNDOLOWAT</td>
<td>int</td>
<td>Minimum amount of data to send for output operations (in bytes).</td>
</tr>
<tr>
<td>SO_SNDTIMEO</td>
<td>struct timeval</td>
<td>Timeout value for a socket send operation.</td>
</tr>
<tr>
<td>SO_TYPE</td>
<td>int</td>
<td>Identify socket type (getsockopt() only).</td>
</tr>
</tbody>
</table>
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_SNDTIMEO option is an option to set a timeout value for the amount of time that an
output function shall block because flow control prevents data from being sent. As noted in
Table 2-2 (on page 64), the option value is a timeval structure with the number of seconds and
microseconds specifying the limit on how long to wait for an output operation to complete. If a
send operation has blocked for this much time, it shall return with a partial count or
errno set to
[EWOULDBLOCK] if no data were sent. The default for this option is the value zero, which
indicates that a send operation will not time out. It is implementation-defined whether the
SO_SNDTIMEO option can be set.

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

### 2.10.17 Use of Sockets for Local UNIX Connections

Support for UNIX domain sockets is mandatory.

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

#### 2.10.17.1 Headers

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

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

### 2.10.18 Use of Sockets over Internet Protocols

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCK_STREAM</td>
<td>IPPROTO_TCP</td>
</tr>
<tr>
<td>SOCK_DGRAM</td>
<td>IPPROTO_UDP</td>
</tr>
<tr>
<td>SOCK_RAW</td>
<td>IPPROTO_RAW</td>
</tr>
<tr>
<td>SOCK_SEQPACKET</td>
<td>Unspecified</td>
</tr>
</tbody>
</table>

A raw interface to IP is available by creating an Internet socket of type SOCK_RAW. The default
protocol for type SOCK_RAW shall be identified in the IP header with the value
IPPROTO_RAW. Applications should not use the default protocol when creating a socket with
type SOCK_RAW, but should identify a specific protocol by value. The ICMP control protocol is
accessible from a raw socket by specifying a value of IPPROTO_ICMP for protocol.
2.10.19 Use of Sockets over Internet Protocols Based on IPv4

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

2.10.19.1 Headers

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

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

2.10.20 Use of Sockets over Internet Protocols Based on IPv6

IPv6

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

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

2.10.20.1 Addressing

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

There are three kinds of IPv6 address:

Unicast

Identifies a single interface.

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

Anycast

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

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

Multicast

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

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

The following special IPv6 addresses are defined:

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

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

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

**IPv4-compatible addresses**
These are assigned to nodes that support IPv6 and can be used when traffic is “tunneled” through IPv4.

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

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

### 2.10.20.2 Compatibility with IPv4

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

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

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

### 2.10.20.3 Interface Identification

Each local interface is assigned a unique positive integer as a numeric index. Indexes start at 1; zero is not used. There may be gaps so that there is no current interface for a particular positive index. Each interface also has a unique implementation-defined name.
The following options apply at the IPPROTO_IPV6 level:

**IPV6_JOIN_GROUP**
When set via `setsockopt()` , it joins the application to a multicast group on an interface (identified by its index) and addressed by a given multicast address, enabling packets sent to that address to be read via the socket. If the interface index is specified as zero, the system selects the interface (for example, by looking up the address in a routing table and using the resulting interface).

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

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

**IPV6_LEAVE_GROUP**
When set via `setsockopt()` , it removes the application from the multicast group on an interface (identified by its index) and addressed by a given multicast address.

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

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

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

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

**IPV6_MULTICAST_IF**
The index of the interface to be used for outgoing multicast packets. It can be set via `setsockopt()` and read via `getsockopt()` . If the interface index is specified as zero, the system selects the interface (for example, by looking up the address in a routing table and using the resulting interface).

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

**IPV6_MULTICAST_LOOP**
This option controls whether outgoing multicast packets should be delivered back to the local application when the sending interface is itself a member of the destination multicast group. If it is set to 1 they are delivered. If it is set to 0 they are not. Other values result in an [EINVAL] error. This option can be set via `setsockopt()` and read via `getsockopt()`.

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

**IPV6_UNICAST_HOPS**
The value of this option is the hop limit for outgoing unicast IPv6 packets sent via the socket. If the option is not set, or is set to −1, the system selects a default value. Attempts to set a value less than −1 or greater than 255 shall result in an [EINVAL] error. This option can be set via `setsockopt()` and read via `getsockopt()`.

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

**IPV6_V6ONLY**
This socket option restricts AF_INET6 sockets to IPv6 communications only. AF_INET6 sockets may be used for both IPv4 and IPv6 communications. Some applications may want to restrict their use of an AF_INET6 socket to IPv6 communications only. For these

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
2855 applications, the IPv6_V6ONLY socket option is defined. When this option is turned on, the
2856 socket can be used to send and receive IPv6 packets only. This is an IPPROTO_IPV6-level
2857 option.
2858 The parameter type of this option is a pointer to an int which is used as a Boolean value.
2859 (Default value: 0)
2860 An [EOPNOTSUPP] error shall result if IPV6_JOIN_GROUP or IPV6_LEAVE_GROUP is used
2861 with getsockopt().

2862 2.10.20.5 Headers
2863 The symbolic constant AF_INET6 is defined in the <sys/socket.h> header to identify the IPv6
2864 Internet address family. See the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13,
2865 Headers.
2866 The sockaddr_storage structure defined in <sys/socket.h> shall be large enough to
2867 accommodate a sockaddr_in6 structure (see the <netinet/in.h> header defined in the Base
2868 Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers) and shall be aligned at an
2869 appropriate boundary so that pointers to it can be cast as pointers to sockaddr_in6 structures
2870 and used to access the fields of those structures without alignment problems. When a
2871 sockaddr_storage structure is cast as a sockaddr_in6 structure, the ss_family field maps onto the
2872 sin6_family field.
2873 The <netinet/in.h>, <arpa/inet.h>, and <netdb.h> headers contain other definitions used in
2874 connection with IPv6 Internet sockets; see the Base Definitions volume of IEEE Std 1003.1-2001,
2875 Chapter 13, Headers.

2876 2.11 Tracing
2877 This section describes extensions to support tracing of user applications. This functionality is
2878 dependent on support of the Trace option (and the rest of this section is not further shaded for
2879 this option).
2880 The tracing facilities defined in IEEE Std 1003.1-2001 allow a process to select a set of trace event
2881 types, to activate a trace stream of the selected trace events as they occur in the flow of
2882 execution, and to retrieve the recorded trace events.
2883 The tracing operation relies on three logically different components: the traced process, the
2884 controller process, and the analyzer process. During the execution of the traced process, when a
2885 trace point is reached, a trace event is recorded into the trace streams created for that process in
2886 which the associated trace event type identifier is not being filtered out. The controller process
2887 controls the operation of recording the trace events into the trace stream. It shall be able to:
2888 • Initialize the attributes of a trace stream
2889 • Create the trace stream (for a specified traced process) using those attributes
2890 • Start and stop tracing for the trace stream
2891 • Filter the type of trace events to be recorded, if the Trace Event Filter option is supported
2892 • Shut a trace stream down
2893 These operations can be done for an active trace stream. The analyzer process retrieves the
2894 traced events either at runtime, when the trace stream has not yet been shut down, but is still
2895 recording trace events; or after opening a trace log that had been previously recorded and shut
2896 down. These three logically different operations can be performed by the same process, or can be
A trace stream identifier can be created by a call to `posix_trace_create()`, `posix_trace_create_withlog()`, or `posix_trace_open()`. The `posix_trace_create()` and `posix_trace_create_withlog()` functions should be used by a controller process. The `posix_trace_open()` should be used by an analyzer process.

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

The events to be traced belong to two classes:

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

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

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

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

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

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

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

### 2.11.1 Tracing Data Definitions

#### 2.11.1.1 Structures

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

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

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

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>posix_stream_status</td>
<td>The operating mode of the trace stream.</td>
</tr>
<tr>
<td>int</td>
<td>posix_stream_full_status</td>
<td>The full status of the trace stream.</td>
</tr>
<tr>
<td>int</td>
<td>posix_stream_overrun_status</td>
<td>Indicates whether trace events were lost in the trace stream.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>posix_stream_flush_status</td>
<td>Indicates whether a flush is in progress.</td>
</tr>
<tr>
<td>int</td>
<td>posix_stream_flush_error</td>
<td>Indicates whether any error occurred during the last flush operation.</td>
</tr>
<tr>
<td>int</td>
<td>posix_log_overrun_status</td>
<td>Indicates whether trace events were lost in the trace log.</td>
</tr>
<tr>
<td>int</td>
<td>posix_log_full_status</td>
<td>The full status of the trace log.</td>
</tr>
</tbody>
</table>

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

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

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

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

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

POSIX_TRACE_NOT_FULL
There is still space available in the trace stream.

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

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

POSIX_TRACE_RUNNING and POSIX_TRACE_FULL
This status combination indicates that tracing is in progress and that the trace stream is full of trace events. This status combination cannot occur unless the stream-full-policy is set to...
POSIX_TRACE_LOOP. The trace stream contains trace events recorded during a moving time window of prior trace events, and some older trace events may have been overwritten and thus lost.

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

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

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

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

POSIX_TRACE_NO_OVERRUN
No trace events were lost.

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

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

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.

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

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

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

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

POSIX_TRACE_NOT_FLUSHING
No flush operation is in progress.

The posix_stream_flush_status member shall be set to POSIX_TRACE_FLUSHING if a flush operation is in progress either due to a call to the posix_trace_flush() function (explicit or caused by a trace stream shutdown operation) or because the trace stream has become full with the stream-full-policy attribute set to POSIX_TRACE_FLUSH. The posix_stream_flush_status member shall be set to POSIX_TRACE_NOT_FLUSHING if no flush operation is in progress.
The `posix_stream_flush_error` member shall be set to zero if no error occurred during flushing. If an error occurred during a previous flushing operation, the `posix_stream_flush_error` member shall be set to the value of the first error that occurred. If more than one error occurs while flushing, error values after the first shall be discarded. The `posix_stream_flush_error` member is reset to zero after its value is read.

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

**POSIX_TRACE_OVERRUN**
- At least one trace event was lost.

**POSIX_TRACE_NO_OVERRUN**
- No trace events were lost.

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

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

**POSIX_TRACE_FULL**
- The space in the trace log is exhausted.

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

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

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

**posix_trace_event_info Structure**

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

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

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

In addition, if the Trace option and the Threads option are both supported, the `posix_trace_event_info` structure defined in `<trace.h>` shall contain the following additional member:
The `posix_event_id` member represents the identification of the trace event type and its value is not directly defined by the user. This identification is returned by a call to one of the following functions: `posix_trace_trid_eventid_open()`, `posix_trace_eventtypelist_getnext_id()`, or `posix_trace_eventid_open()`. The name of the trace event type can be obtained by calling `posix_trace_eventid_get_name()`.

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

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

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

- `POSIX_TRACE_NOT_TRUNCATED`: All the traced data is available.
- `POSIX_TRACE_TRUNCATED_RECORD`: Data was truncated at the time the trace event was generated.
- `POSIX_TRACE_TRUNCATED_READ`: Data was truncated at the time the trace event was read from a trace stream or a trace log because the reader’s buffer was too small. This truncation status overrides the `POSIX_TRACE_TRUNCATED_RECORD` status.

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

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

Otherwise, this member is undefined.

### 2.11.1.2 Trace Stream Attributes

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

- The `generation-version` attribute identifies the origin and version of the trace system.
• The `trace-name` attribute is a character string defined by the trace controller, and that identifies the trace stream.

• The `creation-time` attribute represents the time of the creation of the trace stream.

• The `clock-resolution` attribute defines the clock resolution of the clock used to generate timestamps.

• The `stream-min-size` attribute defines the minimum size in bytes of the trace stream strictly reserved for the trace events.

• The `stream-full-policy` attribute defines the policy followed when the trace stream is full; its value is `POSIX_TRACE_LOOP`, `POSIX_TRACE_UNTIL_FULL`, or `POSIX_TRACE_FLUSH`.

• The `max-data-size` attribute defines the maximum record size in bytes of a trace event.

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

• The `inheritance` attribute specifies whether a newly created trace stream will inherit tracing in its parent’s process trace stream. It is either `POSIX_TRACE_INHERITED` or `POSIX_TRACE_CLOSE_FOR_CHILD`.

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

• If the file type corresponding to the trace log supports the `POSIX_TRACE_LOOP` or the `POSIX_TRACE_UNTIL_FULL` policies, the `log-max-size` attribute defines the maximum size in bytes of the trace log associated with an active trace stream. Other stream data—for example, trace attribute values—shall not be included in this size.

• The `log-full-policy` attribute defines the policy of a trace log associated with an active trace stream to be `POSIX_TRACE_LOOP`, `POSIX_TRACE_UNTIL_FULL`, or `POSIX_TRACE_APPEND`.

### 2.11.2 Trace Event Type Definitions

#### 2.11.2.1 System Trace Event Type Definitions

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

• `POSIX_TRACE_START` shall be associated with a trace start operation.

• `POSIX_TRACE_STOP` shall be associated with a trace stop operation.

• If the Trace Event Filter option is supported, `POSIX_TRACE_FILTER` shall be associated with a trace event type filter change operation.

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

• `POSIX_TRACE_OVERFLOW` shall mark the beginning of a trace overflow condition.

• `POSIX_TRACE_RESUME` shall mark the end of a trace overflow condition.

• If the Trace Log option is supported, `POSIX_TRACE_FLUSH_START` shall mark the beginning of a flush operation.
If the Trace Log option is supported, POSIX_TRACE_FLUSH_STOP shall mark the end of a flush operation.

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

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

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Constant</th>
<th>Associated Data</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>posix_trace_error</code></td>
<td>POSIX_TRACE_ERROR</td>
<td>error</td>
<td>int</td>
</tr>
<tr>
<td><code>posix_trace_start</code></td>
<td>POSIX_TRACE_START</td>
<td>None.</td>
<td></td>
</tr>
<tr>
<td><code>posix_trace_stop</code></td>
<td>POSIX_TRACE_STOP</td>
<td>auto</td>
<td>int</td>
</tr>
<tr>
<td><code>posix_trace_overflow</code></td>
<td>POSIX_TRACE_OVERFLOW</td>
<td>None.</td>
<td></td>
</tr>
<tr>
<td><code>posix_trace_resume</code></td>
<td>POSIX_TRACE_RESUME</td>
<td>None.</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Constant</th>
<th>Associated Data</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>posix_trace_error</td>
<td>POSIX_TRACE_ERROR</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>posix_trace_start</td>
<td>POSIX_TRACE_START</td>
<td>event_filter</td>
<td></td>
</tr>
<tr>
<td>posix_trace_stop</td>
<td>POSIX_TRACE_STOP</td>
<td>auto</td>
<td></td>
</tr>
<tr>
<td>posix_trace_filter</td>
<td>POSIX_TRACE_FILTER</td>
<td>old_event_filter</td>
<td></td>
</tr>
<tr>
<td>posix_trace_overflow</td>
<td>POSIX_TRACE_OVERFLOW</td>
<td>None.</td>
<td></td>
</tr>
<tr>
<td>posix_trace_resume</td>
<td>POSIX_TRACE_RESUME</td>
<td>None.</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Constant</th>
<th>Associated Data</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>posix_trace_error</td>
<td>POSIX_TRACE_ERROR</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>posix_trace_start</td>
<td>POSIX_TRACE_START</td>
<td>None.</td>
<td></td>
</tr>
<tr>
<td>posix_trace_stop</td>
<td>POSIX_TRACE_STOP</td>
<td>auto</td>
<td></td>
</tr>
<tr>
<td>posix_trace_overflow</td>
<td>POSIX_TRACE_OVERFLOW</td>
<td>None.</td>
<td></td>
</tr>
<tr>
<td>posix_trace_resume</td>
<td>POSIX_TRACE_RESUME</td>
<td>None.</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Constant</th>
<th>Associated Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>posix_trace_error</td>
<td>POSIX_TRACE_ERROR</td>
<td>error</td>
</tr>
<tr>
<td>posix_trace_start</td>
<td>POSIX_TRACE_START</td>
<td>event_filter</td>
</tr>
<tr>
<td>posix_trace_stop</td>
<td>POSIX_TRACE_STOP</td>
<td>auto</td>
</tr>
<tr>
<td>posix_trace_filter</td>
<td>POSIX_TRACE_FILTER</td>
<td>old_event_filter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>new_event_filter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trace_event_set_t</td>
</tr>
<tr>
<td>posix_trace_overflow</td>
<td>POSIX_TRACE_OVERFLOW</td>
<td>None.</td>
</tr>
<tr>
<td>posix_trace_resume</td>
<td>POSIX_TRACE_RESUME</td>
<td>None.</td>
</tr>
<tr>
<td>posix_trace_flush_start</td>
<td>POSIX_TRACE_FLUSH_START</td>
<td>None.</td>
</tr>
<tr>
<td>posix_trace_flush_stop</td>
<td>POSIX_TRACE_FLUSH_STOP</td>
<td>None.</td>
</tr>
</tbody>
</table>

#### 2.11.2.2 User Trace Event Type Definitions

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

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

#### Table 2-7 Trace Option: User Trace Event

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>posix_trace_unnamed_userevent</td>
<td>POSIX_TRACE_UNNAMED_USEREVENT</td>
</tr>
</tbody>
</table>

#### 2.11.3 Trace Functions

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

- A trace controller process requires functions to set up and customize all the resources needed to run a trace stream, including:
  - Attribute initialization and destruction: `posix_trace_attr_init()`
  - Identification information manipulation: `posix_trace_attr_getgenversion()`
  - Trace system behavior modification: `posix_trace_attr_getinherited()`
  - Trace stream and trace log size set: `posix_trace_attr_getmaxusereventsize()`
— Trace stream creation, flush, and shutdown \((\text{posix\_trace\_create()})\)
— Trace stream and trace log clear \((\text{posix\_trace\_clear()})\)
— Trace event type identifier manipulation \((\text{posix\_trace\_trid\_eventid\_open()})\)
— Trace event type identifier list exploration \((\text{posix\_trace\_eventtypelist\_getnext\_id()})\)
— Trace event type set manipulation \((\text{posix\_trace\_eventset\_empty()})\)
— Trace event type filter set \((\text{posix\_trace\_set\_filter()})\)
— Trace stream start and stop \((\text{posix\_trace\_start()})\)
— Trace stream information and status read \((\text{posix\_trace\_get\_attr()})\)

• A traced process requires functions to instrument trace points:
— Trace event type identifiers definition and trace points insertion \((\text{posix\_trace\_event()})\)

• A trace analyzer process requires functions to retrieve information from a trace stream and trace log:
— Identification information read \((\text{posix\_trace\_attr\_getgenversion()})\)
— Trace system behavior information read \((\text{posix\_trace\_attr\_getinherited()})\)
— Trace stream and trace log size get \((\text{posix\_trace\_attr\_getmaxusereventsizeresize()})\)
— Trace event type identifier manipulation \((\text{posix\_trace\_trid\_eventid\_open()})\)
— Trace event type identifier list exploration \((\text{posix\_trace\_eventtypelist\_getnext\_id()})\)
— Trace log open, rewind, and close \((\text{posix\_trace\_open()})\)
— Trace stream information and status read \((\text{posix\_trace\_get\_attr()})\)
— Trace event read \((\text{posix\_trace\_getnext\_event()})\)

### 2.12 Data Types

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

<table>
<thead>
<tr>
<th>Defined Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc_t</td>
<td>Type used for terminal special characters.</td>
</tr>
<tr>
<td>clock_t</td>
<td>Integer or real-floating type used for processor times, as defined in the ISO C standard.</td>
</tr>
<tr>
<td>clockid_t</td>
<td>Used for clock ID type in some timer functions.</td>
</tr>
<tr>
<td>dev_t</td>
<td>Arithmetic type used for device numbers.</td>
</tr>
<tr>
<td>DIR</td>
<td>Type representing a directory stream.</td>
</tr>
<tr>
<td>div_t</td>
<td>Structure type returned by the (\text{div()}) function.</td>
</tr>
<tr>
<td>FILE</td>
<td>Structure containing information about a file.</td>
</tr>
<tr>
<td>glob_t</td>
<td>Structure type used in pathname pattern matching.</td>
</tr>
<tr>
<td>fpos_t</td>
<td>Type containing all information needed to specify uniquely every line.</td>
</tr>
<tr>
<td>Defined Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>gid_t</td>
<td>Integer type used for group IDs.</td>
</tr>
<tr>
<td>iconv_t</td>
<td>Type used for conversion descriptors.</td>
</tr>
<tr>
<td>id_t</td>
<td>Integer type used as a general identifier; can be used to contain</td>
</tr>
<tr>
<td></td>
<td>at least the largest of a pid_t, uid_t, or gid_t.</td>
</tr>
<tr>
<td>ino_t</td>
<td>Unsigned integer type used for file serial numbers.</td>
</tr>
<tr>
<td>key_t</td>
<td>Arithmetic type used for XSI interprocess communication.</td>
</tr>
<tr>
<td>ldiv_t</td>
<td>Structure type returned by the ldiv() function.</td>
</tr>
<tr>
<td>mode_t</td>
<td>Integer type used for file attributes.</td>
</tr>
<tr>
<td>mqd_t</td>
<td>Used for message queue descriptors.</td>
</tr>
<tr>
<td>nfds_t</td>
<td>Integer type used for the number of file descriptors.</td>
</tr>
<tr>
<td>nlink_t</td>
<td>Integer type used for link counts.</td>
</tr>
<tr>
<td>off_t</td>
<td>Signed integer type used for file sizes.</td>
</tr>
<tr>
<td>pid_t</td>
<td>Signed integer type used for process and process group IDs.</td>
</tr>
<tr>
<td>pthread_attr_t</td>
<td>Used to identify a thread attribute object.</td>
</tr>
<tr>
<td>pthread_cond_t</td>
<td>Used for condition variables.</td>
</tr>
<tr>
<td>pthread_condattr_t</td>
<td>Used to identify a condition attribute object.</td>
</tr>
<tr>
<td>pthread_key_t</td>
<td>Used for thread-specific data keys.</td>
</tr>
<tr>
<td>pthread_mutex_t</td>
<td>Used for mutexes.</td>
</tr>
<tr>
<td>pthread_mutexattr_t</td>
<td>Used to identify a mutex attribute object.</td>
</tr>
<tr>
<td>pthread_once_t</td>
<td>Used for dynamic package initialization.</td>
</tr>
<tr>
<td>pthread_rwlock_t</td>
<td>Used for read-write locks.</td>
</tr>
<tr>
<td>pthread_rwlockattr_t</td>
<td>Used for read-write lock attributes.</td>
</tr>
<tr>
<td>pthread_t</td>
<td>Used to identify a thread.</td>
</tr>
<tr>
<td>ptdiff_t</td>
<td>Signed integer type of the result of subtracting two pointers.</td>
</tr>
<tr>
<td>reg_t</td>
<td>Structure type used in regular expression matching.</td>
</tr>
<tr>
<td>regmatch_t</td>
<td>Structure type used in regular expression matching.</td>
</tr>
<tr>
<td>rlim_t</td>
<td>Unsigned integer type used for limit values, to which objects of type</td>
</tr>
<tr>
<td></td>
<td>int and off_t can be cast without loss of value.</td>
</tr>
<tr>
<td>sem_t</td>
<td>Type used in performing semaphore operations.</td>
</tr>
<tr>
<td>sig_atomic_t</td>
<td>Integer type of an object that can be accessed as an atomic entity, even</td>
</tr>
<tr>
<td></td>
<td>in the presence of asynchronous interrupts.</td>
</tr>
<tr>
<td>sigset_t</td>
<td>Integer or structure type of an object used to represent sets of signals.</td>
</tr>
<tr>
<td>size_t</td>
<td>Unsigned integer type used for size of objects.</td>
</tr>
<tr>
<td>speed_t</td>
<td>Type used for terminal baud rates.</td>
</tr>
<tr>
<td>ssize_t</td>
<td>Signed integer type used for a count of bytes or an error indication.</td>
</tr>
<tr>
<td>suseconds_t</td>
<td>Signed integer type used for time in microseconds.</td>
</tr>
<tr>
<td>tcfiag_t</td>
<td>Type used for terminal modes.</td>
</tr>
<tr>
<td>time_t</td>
<td>Integer or real-floating type used for time in seconds, as defined in the</td>
</tr>
<tr>
<td></td>
<td>ISO C standard.</td>
</tr>
<tr>
<td>timer_t</td>
<td>Used for timer ID returned by the timer_create() function.</td>
</tr>
<tr>
<td>uid_t</td>
<td>Integer type used for user IDs.</td>
</tr>
<tr>
<td>useconds_t</td>
<td>Unsigned integer type used for time in microseconds.</td>
</tr>
<tr>
<td>va_list</td>
<td>Type used for traversing variable argument lists.</td>
</tr>
<tr>
<td>wchar_t</td>
<td>Integer type whose range of values can represent distinct codes for</td>
</tr>
<tr>
<td>Defined Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>wctype_t</td>
<td>Scalar type which represents a character class descriptor.</td>
</tr>
<tr>
<td>wint_t</td>
<td>Integer type capable of storing any valid value of wchar_t or WEOF.</td>
</tr>
<tr>
<td>wordexp_t</td>
<td>Structure type used in word expansion.</td>
</tr>
</tbody>
</table>
This chapter describes the functions, macros, and external variables to support applications portability at the C-language source level.
FD_CLR() — macros for synchronous I/O multiplexing

SYNOPSIS

```c
#include <sys/time.h>

FD_CLR(int fd, fd_set *fdset);
FD_ISSET(int fd, fd_set *fdset);
FD_SET(int fd, fd_set *fdset);
FD_ZERO(fd_set *fdset);
```

DESCRIPTION

Refer to `pselect()`.
NAME
.Exit, _exit — terminate a process

SYNOPSIS
#include <unistd.h>

void _Exit(int status);
void _exit(int status);

DESCRIPTION
Refer to exit().
NAME

_longjmp, _setjmp — non-local goto

SYNOPSIS

xsi

```c
#include <setjmp.h>

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

DESCRIPTION

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.

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

RETURN VALUE

Refer to longjmp() and setjmp().

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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

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

The _longjmp() and _setjmp() functions are included to support programs written to historical system interfaces. New applications should use siglongjmp() and sigset jmp() respectively.

RATIONALE

None.

FUTURE DIRECTIONS

The _longjmp() and _set jmp() functions may be marked LEGACY in a future version.

SEE ALSO

longjmp(), set jmp(), siglongjmp(), sigset jmp(), the Base Definitions volume of IEEE Std 1003.1-2001, <set jmp.h>

CHANGE HISTORY

First released in Issue 4, Version 2.
3419  **Issue 5**
3420     Moved from X/OPEN UNIX extension to BASE.
NAME
_tolower — transliterate uppercase characters to lowercase

SYNOPSIS
#include <ctype.h>

int _tolower(int c);

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

RETURN VALUE
Upon successful completion, _tolower() shall return the lowercase letter corresponding to the
argument passed.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

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

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
_toupper — transliterate lowercase characters to uppercase

SYNOPSIS

XSI
#include <ctype.h>

int _toupper(int c);

DESCRIPTION

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

RETURN VALUE

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

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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

CHANGE HISTORY

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

Issue 6

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
a64l, l64a — convert between a 32-bit integer and a radix-64 ASCII string

SYNOPSIS
XSI
#include <stdlib.h>
long a64l(const char *s);
char *l64a(long value);

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

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

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

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

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

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

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

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

ERRORS
No errors are defined.

EXAMPLES
None.

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

RATIONALE
This is not the same encoding as used by either encoding variant of the uuencode utility.
FUTURE DIRECTIONS
None.

SEE ALSO
strtol(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>, the Shell and Utilities volume of IEEE Std 1003.1-2001, uuencode

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.
A note indicating that these functions need not be reentrant is added to the DESCRIPTION.
NAME
abort — generate an abnormal process abort

SYNOPSIS
#include <stdlib.h>
void abort(void);

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

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

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

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

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

RETURN VALUE
The abort() function shall not return.

ERRORS
No errors are defined.

EXAMPLES
None.

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

RATIONALE
None.

FUTURE DIRECTIONS
None.

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

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.
NAME
abs — return an integer absolute value

SYNOPSIS
#include <stdlib.h>
int abs(int i);

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

RETURN VALUE
The abs() function shall return the absolute value of its integer operand.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
In two's-complement representation, the absolute value of the negative integer with largest magnitude [INT_MIN] might not be representable.

RATIONALE
None.

FUTURE DIRECTIONS
None.

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

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.
NAME
accept — accept a new connection on a socket

SYNOPSIS
#include <sys/socket.h>

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

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

The accept() function takes the following arguments:

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

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

descriptor_len Points to a socklen_t structure which on input specifies the length of the
    supplied sockaddr structure, and on output specifies the length of the stored
    address.

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

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

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

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

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

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

ERRORS
The accept() function shall fail if:

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

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

[ECONNABORTED]
    A connection has been aborted.
The `accept()` function was interrupted by a signal that was caught before a valid connection arrived.

The `socket` is not accepting connections.

`[OPEN_MAX]` file descriptors are currently open in the calling process.

The maximum number of file descriptors in the system are already open.

The `socket` argument does not refer to a socket.

The `socket` type of the specified socket does not support accepting connections.

The `accept()` function may fail if:

- `ENOBUFS` No buffer space is available.
- `ENOMEM` There was insufficient memory available to complete the operation.
- `EPROTO` A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized.

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

None.

None.

None.

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

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

The `restrict` keyword is added to the `accept()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
access — determine accessibility of a file

SYNOPSIS
#include <unistd.h>

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

DESCRIPTION
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.

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).

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

RETURN VALUE
If the requested access is permitted, access() succeeds and shall return 0; otherwise, −1 shall be returned and errno shall be set to indicate the error.

ERRORS
The access() function shall fail if:

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

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

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

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

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

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

The access() function may fail if:

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

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

[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}.

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

Testing for the Existence of a File

The following example tests whether a file named _myfile_ exists in the `/tmp` directory.

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

APPLICATION USAGE

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

RATIONALE

In early proposals, some inadequacies in the _access()_ function led to the creation of an _eaccess()_ function because:

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

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

The sentence concerning appropriate privileges and execute permission bits reflects the two possibilities implemented by historical implementations when checking superuser access for _X_OK_.

New implementations are discouraged from returning _X_OK_ unless at least one execution permission bit is set.

FUTURE DIRECTIONS

None.

SEE ALSO

_chmod(), stat(),_ the Base Definitions volume of IEEE Std 1003.1-2001, _<unistd.h>_

CHANGE HISTORY

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

Issue 6

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

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

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

• The [ELOOP] optional error condition is added.
NAME
acos, acosf, acosl — arc cosine functions

SYNOPSIS
#include <math.h>
double acos(double x);
float acosf(float x);
long double acosl(long double x);

DESCRIPTION
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

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

An application wishing to check for error situations should set errno to zero and call
fclearexcept(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.

RETURN VALUE
Upon successful completion, these functions shall return the arc cosine of x, in the range [0,\pi]
radians.

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.

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

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

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

ERRORS
These functions shall fail if:

MX Domain Error The x argument is finite and is not in the range [-1,1], or is ±\Inf.

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

EXAMPLES
None.

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

RATIONALE
None.
acos()  

FUTURE DIRECTIONS
None.

SEE ALSO

cos(), 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>

CHANGE HISTORY

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

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The acosf() and acosl() functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.
NAME
acosh, acoshf, acoshl, — inverse hyperbolic cosine functions

SYNOPSIS
#include <math.h>
double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);

DESCRIPTION
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.

These functions shall compute the inverse hyperbolic cosine of their argument \( x \).

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

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

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

If \( x \) is NaN, a NaN shall be returned.

If \( x \) is +1, +0 shall be returned.

If \( x \) is +Inf, +Inf shall be returned.

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

ERRORS
These functions shall fail if:

Domain Error
The \( x \) argument is finite and less than +1.0, or is −Inf.

If the integer expression (math_errno & MATH_ERRNO) is non-zero, then \( \text{errno} \) shall be set to [EDOM]. If the integer expression (math_errno & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

EXAMPLES
None.

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

RATIONALE
None.
FUTURE DIRECTIONS
None.

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

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
The acosh() function is no longer marked as an extension.
The acoshf(), and acoshl() functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.
NAME
acosl — arc cosine functions

SYNOPSIS
#include <math.h>
long double acosl(long double x);

DESCRIPTION
Refer to acos().
NAME
aio_cancel — cancel an asynchronous I/O request (REALTIME)

SYNOPSIS
#include <aio.h>

int aio_cancel(int fildes, struct aiocb *aiocbp);

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

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

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

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

Which operations are cancelable is implementation-defined.

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

ERRORS
The aio_cancel() function shall fail if:

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

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

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO

\texttt{aio\_read()}, \texttt{aio\_write()}, the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<aio.h>}

CHANGE HISTORY

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

Issue 6

The \texttt{[ENOSYS]} error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.
NAME
aio_error — retrieve errors status for an asynchronous I/O operation (REALTIME)

SYNOPSIS
AIO
#include <aio.h>

int aio_error(const struct aiocb *aiocbp);

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

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

ERRORS
The aio_error() function may fail if:

[EINVAL] The aiocbp argument does not refer to an asynchronous operation whose
return status has not yet been retrieved.

EXAMPLES
None.

APPLICATION USAGE
The aio_error() function is part of the Asynchronous Input and Output option and need not be
available on all implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
aio_cancel(), aio_fsync(), aio_read(), aio_return(), aio_write(), close(), exec, exit(), fork(), lio_listio(),
seek(), read(), the Base Definitions volume of IEEE Std 1003.1-2001, <aio.h>

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

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

The APPLICATION USAGE section is added.
NAME

aio_fsync — asynchronous file synchronization (REALTIME)

SYNOPSIS

AIO

```c
#include <aio.h>
int aio_fsync(int op, struct aiocb *aiocbp);
```

DESCRIPTION

The `aio_fsync()` function shall asynchronously force all I/O operations associated with the file indicated by the file descriptor `aio_fildes` member of the `aiocb` structure referenced by the `aiocbp` argument and queued at the time of the call to `aio_fsync()` to the synchronized I/O completion state. The function call shall return when the synchronization request has been initiated or queued to the file or device (even when the data cannot be synchronized immediately).

If `op` is `O_DSYNC`, all currently queued I/O operations shall be completed as if by a call to `fdatasync()`; that is, as defined for synchronized I/O data integrity completion. If `op` is `O_SYNC`, all currently queued I/O operations shall be completed as if by a call to `fsync()`; that is, as defined for synchronized I/O file integrity completion. If the `aio_fsync()` function fails, or if the operation queued by `aio_fsync()` fails, then, as for `fsync()` and `fdatasync()`, outstanding I/O operations are not guaranteed to have been completed.

If `aio_fsync()` succeeds, then it is only the I/O that was queued at the time of the call to `aio_fsync()` that is guaranteed to be forced to the relevant completion state. The completion of subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized fashion.

The `aiocbp` argument refers to an asynchronous I/O control block. The `aiocbp` value may be used as an argument to `aio_error()` and `aio_return()` in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. When the request is queued, the error status for the operation is [EINPROGRESS]. When all data has been successfully transferred, the error status shall be reset to reflect the success or failure of the operation. If the operation does not complete successfully, the error status for the operation shall be set to indicate the error. The `aio_sigevent` member determines the asynchronous notification to occur as specified in Section 2.4.1 (on page 28) when all operations have achieved synchronized I/O completion. All other members of the structure referenced by `aiocbp` are ignored. If the control block referenced by `aiocbp` becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

If the `aio_fsync()` function fails or `aiocbp` indicates an error condition, data is not guaranteed to have been successfully transferred.

RETURN VALUE

The `aio_fsync()` function shall return the value 0 to the calling process if the I/O operation is successfully queued; otherwise, the function shall return the value −1 and set `errno` to indicate the error.

ERRORS

The `aio_fsync()` function shall fail if:

- **[EAGAIN]** The requested asynchronous operation was not queued due to temporary resource limitations.
- **[EBADF]** The `aio_fildes` member of the `aiocb` structure referenced by the `aiocbp` argument is not a valid file descriptor open for writing.
This implementation does not support synchronized I/O for this file.

A value of op other than O_DSYNC or O_SYNC was specified.

In the event that any of the queued I/O operations fail, aio_fsync() shall return the error condition defined for read() and write(). The error is returned in the error status for the asynchronous fsync() operation, which can be retrieved using aio_error().
NAME
aio_read — asynchronous read from a file (REALTIME)

SYNOPSIS
AIO
#include <aio.h>

int aio_read(struct aiocb *aiocbp);

DESCRIPTION
The aio_read() function shall read aios->aio_nbytes from the file associated with
aiocbp->aio_fildes into the buffer pointed to by aios->aio_buf. The function call shall return when
the read request has been initiated or queued to the file or device (even when the data cannot be
delivered immediately).

If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted
at a priority equal to the scheduling priority of the process minus aios->aio_reqprio.

The aios value may be used as an argument to aio_error() and aio_return() in order to
determine the error status and return status, respectively, of the asynchronous operation while it
is proceeding. If an error condition is encountered during queuing, the function call shall return
without having initiated or queued the request. The requested operation takes place at the
absolute position in the file as given by aios_offset, as if lseek() were called immediately prior to
the operation with an offset equal to aios_offset and a whence equal to SEEK_SET. After a
successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file
is unspecified.

The aios->lio_opcode field shall be ignored by aio_read().

The aios argument points to an aiocb structure. If the buffer pointed to by aios->aio_buf or
the control block pointed to by aios becomes an illegal address prior to asynchronous I/O
completion, then the behavior is undefined.

Simultaneous asynchronous operations using the same aios produce undefined results.

If synchronized I/O is enabled on the file associated with aios->aio_fildes, the behavior of this
function shall be according to the definitions of synchronized I/O data integrity completion and
synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is
outstanding to the address range being changed, the result of that action is undefined.

For regular files, no data transfer shall occur past the offset maximum established in the open
file description associated with aios->aio_fildes.

RETURN VALUE
The aio_read() function shall return the value zero to the calling process if the I/O operation is
successfully queued; otherwise, the function shall return the value −1 and set errno to indicate
the error.

ERRORS
The aio_read() function shall fail if:

[EAGAIN] The requested asynchronous I/O operation was not queued due to system
resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to
aio_read(), or asynchronously. If any of the conditions below are detected synchronously, the
aio_read() function shall return −1 and set errno to the corresponding value. If any of the
conditions below are detected asynchronously, the return status of the asynchronous operation
is set to −1, and the error status of the asynchronous operation is set to the corresponding value.

[EBADF] The aiocbp->aio_fildes argument is not a valid file descriptor open for reading.

[EINVAL] The file offset value implied by aiocbp->aio_offset would be invalid, aiocbp->aio_reqprio is not a valid value, or aiocbp->aio_nbytes is an invalid value.

In the case that the aio_read() successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is one of the values normally returned by the read() function call. In addition, the error status of the asynchronous operation is set to one of the error statuses normally set by the read() function call, or one of the following values:

[EBADF] The aiocbp->aio_fildes argument is not a valid file descriptor open for reading.

[ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit aio_cancel() request.

[EINVAL] The file offset value implied by aiocbp->aio_offset would be invalid.

The following condition may be detected synchronously or asynchronously:

[E.overflow] The file is a regular file, aiocbp->aio_nbytes is greater than 0, and the starting offset in aiocbp->aio_offset is before the end-of-file and is at or beyond the offset maximum in the open file description associated with aiocbp->aio_fildes.

EXAMPLES
None.

APPLICATION USAGE
The aio_read() function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
aio_cancel(), aio_error(), lio_listio(), aio_return(), aio_write(), close(), exec, exit(), fork(), lseek(), read(), the Base Definitions volume of IEEE Std 1003.1-2001, <aio.h>

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

Large File Summit extensions are added.

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

The APPLICATION USAGE section is added.
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.
NAME
aio_return — retrieve return status of an asynchronous I/O operation (REALTIME)

SYNOPSIS
#include <aio.h>

ssize_t aio_return(struct aiocb *aiocbp);

DESCRIPTION
The aio_return() function shall return the return status associated with the aiocb structure referenced by the aiocbp argument. The return status for an asynchronous I/O operation is the value that would be returned by the corresponding read(), write(), or fsync() function call. If the error status for the operation is equal to [EINPROGRESS], then the return status for the operation is undefined. The aio_return() function may be called exactly once to retrieve the return status of a given asynchronous operation; thereafter, if the same aiocb structure is used in a call to aio_return() or aio_error(), an error may be returned. When the aiocb structure referred to by aiocbp is used to submit another asynchronous operation, then aio_return() may be successfully used to retrieve the return status of that operation.

RETURN VALUE
If the asynchronous I/O operation has completed, then the return status, as described for read(), write(), and fsync(), shall be returned. If the asynchronous I/O operation has not yet completed, the results of aio_return() are undefined.

ERRORS
The aio_return() function may fail if:

[EINVAL] The aiocbp argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

EXAMPLES
None.

APPLICATION USAGE
The aio_return() function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
aio_cancel(), aio_error(), aio_fsync(), aio_read(), aio_write(), close(), exec(), exit(), fork(), flio_listio(), lseek(), read(), the Base Definitions volume of IEEE Std 1003.1-2001, <aio.h>

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

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

The APPLICATION USAGE section is added.

The [EINVAL] error condition is updated as a “may fail”. This is for consistency with the DESCRIPTION.
NAME
aio_suspend — wait for an asynchronous I/O request (REALTIME)

SYNOPSIS
AIO #include <aio.h>

int aio_suspend(const struct aiocb * const list[], int nent,
const struct timespec *timeout);

DESCRIPTION
The aio_suspend() function shall suspend the calling thread until at least one of the asynchronous I/O operations referenced by the list argument has completed, until a signal interrupts the function, or, if timeout is not NULL, until the time interval specified by timeout has passed. If any of the aiocb structures in the list correspond to completed asynchronous I/O operations (that is, the error status for the operation is not equal to [EINPROGRESS]) at the time of the call, the function shall return without suspending the calling thread. The list argument is an array of pointers to asynchronous I/O control blocks. The nent argument indicates the number of elements in the array. Each aiocb structure pointed to has been used in initiating an asynchronous I/O request via aio_read(), aio_write(), or lio_listio(). This array may contain NULL pointers, which are ignored. If this array contains pointers that refer to aiocb structures that have not been used in submitting asynchronous I/O, the effect is undefined.

If the time interval indicated in the timespec structure pointed to by timeout passes before any of the I/O operations referenced by list are completed, then aio_suspend() shall return with an error. If the Monotonic Clock option is supported, the clock that shall be used to measure this time interval shall be the CLOCK_MONOTONIC clock.

RETURN VALUE
If the aio_suspend() function returns after one or more asynchronous I/O operations have completed, the function shall return zero. Otherwise, the function shall return a value of −1 and set errno to indicate the error.

The application may determine which asynchronous I/O completed by scanning the associated error and return status using aio_error() and aio_return(), respectively.

ERRORS
The aio_suspend() function shall fail if:

[EAGAIN] No asynchronous I/O indicated in the list referenced by list completed in the time interval indicated by timeout.

[EINTR] A signal interrupted the aio_suspend() function. Note that, since each asynchronous I/O operation 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.

EXAMPLES
None.

APPLICATION USAGE
The aio_suspend() function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE
None.
FUTURE DIRECTIONS

None.

SEE ALSO

aio_read(), aio_write(), lio_listio(), the Base Definitions volume of IEEE Std 1003.1-2001, <aio.h>

CHANGE HISTORY

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

Issue 6

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

The APPLICATION USAGE section is added.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the CLOCK_MONOTONIC clock, if supported, is used.
NAME
aio_write — asynchronous write to a file (REALTIME)

SYNOPSIS
AIO
#include <aio.h>

int aio_write(struct aiocb *aiocbp);

DESCRIPTION
The aio_write() function shall write aiocbp->aio_nbytes to the file associated with aiocbp->aio_fildes from the buffer pointed to by aiocbp->aio_buf. The function shall return when the write request has been initiated or, at a minimum, queued to the file or device.

If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted at a priority equal to the scheduling priority of the process minus aiocbp->aio_reqprio.

The aiocbp argument may be used as an argument to aio_error() and aio_return() in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding.

The aiocbp argument points to an aiocb structure. If the buffer pointed to by aiocbp->aio_buf or the control block pointed to by aiocbp becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

If O_APPEND is not set for the file descriptor aiocbp->aio_fildes, then the requested operation shall take place at the absolute position in the file as given by aio_offset, as if lseek() were called immediately prior to the operation with an offset equal to aio_offset and a whence equal to SEEK_SET. If O_APPEND is set for the file descriptor, write operations append to the file in the same order as the calls were made. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The aiocbp->aio_lio_opcode field shall be ignored by aio_write().

Simultaneous asynchronous operations using the same aiocbp produce undefined results.

If synchronized I/O is enabled on the file associated with aiocbp->aio_fildes, the behavior of this function shall be according to the definitions of synchronized I/O data integrity completion, and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding to the address range being changed, the result of that action is undefined.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with aiocbp->aio_fildes.

RETURN VALUE
The aio_write() function shall return the value zero to the calling process if the I/O operation is successfully queued; otherwise, the function shall return the value −1 and set errno to indicate the error.

ERRORS
The aio_write() function shall fail if:

[EAGAIN] The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to aio_write(), or asynchronously. If any of the conditions below are detected synchronously, the aio_write() function shall return −1 and set errno to the corresponding value. If any of the
conditions below are detected asynchronously, the return status of the asynchronous operation shall be set to −1, and the error status of the asynchronous operation is set to the corresponding value.

- **[EBADF]** The aiocbp->aio_fildes argument is not a valid file descriptor open for writing.
- **[EINVAL]** The file offset value implied by aiocbp->aio_offset would be invalid, aiocbp->aio_reqprio is not a valid value, or aiocbp->aio_nbytes is an invalid value.

In the case that the aio_write() successfully queues the I/O operation, the return status of the asynchronous operation shall be one of the values normally returned by the write() function call. If the operation is successfully queued but is subsequently canceled or encounters an error, the error status for the asynchronous operation contains one of the values normally set by the write() function call, or one of the following:

- **[EBADF]** The aiocbp->aio_fildes argument is not a valid file descriptor open for writing.
- **[EINVAL]** The file offset value implied by aiocbp->aio_offset would be invalid.
- **[ECANCELED]** The requested I/O was canceled before the I/O completed due to an explicit aio_cancel() request.

The following condition may be detected synchronously or asynchronously:

- **[EFBIG]** The file is a regular file, aiocbp->aio_nbytes is greater than 0, and the starting offset in aiocbp->aio_offset is at or beyond the offset maximum in the open file description associated with aiocbp->aio_fildes.

**EXAMPLES**

None.

**APPLICATION USAGE**

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

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

aio_cancel(), aio_error(), aio_read(), aio_return(), close(), exec, exit(), fork(), lio_listio(), lseek(), write(), the Base Definitions volume of IEEE Std 1003.1-2001, <aio.h>

**CHANGE HISTORY**

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

Large File Summit extensions are added.

**Issue 6**

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

The APPLICATION USAGE section is added.

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

- In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs past the offset maximum established in the open file description associated with
aio_write( ), \texttt{aiocbp->aio_fildes}.

- The [EBIG] error is added as part of the large file support extensions.
alarm()  System Interfaces

NAME
alarm — schedule an alarm signal

SYNOPSIS
#include <unistd.h>

unsigned alarm(unsigned seconds);

DESCRIPTION
The alarm() function shall cause the system to generate a SIGALRM signal for the process after
the number of realtime seconds specified by seconds have elapsed. Processor scheduling delays
may prevent the process from handling the signal as soon as it is generated.

If seconds is 0, a pending alarm request, if any, is canceled.

Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner.
If the SIGALRM signal has not yet been generated, the call shall result in rescheduling the time
at which the SIGALRM signal is generated.

XSI Interactions between alarm() and any of setitimer(), ualarm(), or usleep() are unspecified.

RETURN VALUE
If there is a previous alarm() request with time remaining, alarm() shall return a non-zero value
that is the number of seconds until the previous request would have generated a SIGALRM
signal. Otherwise, alarm() shall return 0.

ERRORS
The alarm() function is always successful, and no return value is reserved to indicate an error.

EXAMPLES
None.

APPLICATION USAGE
The fork() function clears pending alarms in the child process. A new process image created by
one of the exec functions inherits the time left to an alarm signal in the old process' image.

Application writers should note that the type of the argument seconds and the return value of
alarm() is unsigned. That means that a Strictly Conforming POSIX System Interfaces
Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX},
which the ISO C standard sets as 65535, and any application passing a larger value is restricting
its portability. A different type was considered, but historical implementations, including those
with a 16-bit int type, consistently use either unsigned or int.

Application writers should be aware of possible interactions when the same process uses both
the alarm() and sleep() functions.

RATIONALE
Many historical implementations (including Version 7 and System V) allow an alarm to occur up
to a second early. Other implementations allow alarms up to half a second or one clock tick
early or do not allow them to occur early at all. The latter is considered most appropriate, since it
gives the most predictable behavior, especially since the signal can always be delayed for an
indefinite amount of time due to scheduling. Applications can thus choose the seconds argument
as the minimum amount of time they wish to have elapse before the signal.

The term "realtime" here and elsewhere (sleep(), times()) is intended to mean "wall clock" time
as common English usage, and has nothing to do with "realtime operating systems". It is in
contrast to virtual time, which could be misinterpreted if just time were used.

In some implementations, including 4.3 BSD, very large values of the seconds argument are
silently rounded down to an implementation-defined maximum value. This maximum is large
There were two possible choices for alarm generation in multi-threaded applications: generation for the calling thread or generation for the process. The first option would not have been particularly useful since the alarm state is maintained on a per-process basis and the alarm that is established by the last invocation of `alarm()` is the only one that would be active.

Furthermore, allowing generation of an asynchronous signal for a thread would have introduced an exception to the overall signal model. This requires a compelling reason in order to be justified.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
alarm(), exec, fork(), getitimer(), pause(), sigaction(), sleep(), ualarm(), usleep(), the Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`, `<unistd.h>`

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 6**
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to indicate that interactions with the `setitimer()`, `ualarm()`, and `usleep()` functions are unspecified.
NAME
asctime, asctime_r — convert date and time to a string

SYNOPSIS
#include <time.h>

char *asctime(const struct tm *timeptr);
TSF char *asctime_r(const struct tm *restrict tm, char *restrict buf);

DESCRIPTION
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:
Sun Sep 16 01:03:52 1973
using the equivalent of the following algorithm:
char *asctime(const struct tm *timeptr)
{
static char wday_name[7][3] = {
  "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
};
static char mon_name[12][3] = {
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};
static char result[26];

sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
  wday_name[timeptr->tm_wday],
  mon_name[timeptr->tm_mon],
  timeptr->tm_mday, timeptr->tm_hour,
  timeptr->tm_min, timeptr->tm_sec,
  1900 + timeptr->tm_year);
return result;
}

The tm structure is defined in the <time.h> header.

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.

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.
RETURN VALUE
Upon successful completion, `asctime()` shall return a pointer to the string.

Upon successful completion, `asctime_r()` shall return a pointer to a character string containing the date and time. This string is pointed to by the argument `buf`. If the function is unsuccessful, it shall return NULL.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
Values for the broken-down time structure can be obtained by calling `gmtime()` or `localtime()`.
This function is included for compatibility with older implementations, and does not support localized date and time formats. Applications should use `strftime()` to achieve maximum portability.

The `asctime_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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
`clock()`, `ctime()`, `difftime()`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `strptime()`, `time()`, `utime()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<time.h>`

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.
The `asctime_r()` function is included for alignment with the POSIX Threads Extension.
A note indicating that the `asctime()` function need not be reentrant is added to the DESCRIPTION.

Issue 6
The `asctime_r()` function is marked as part of the Thread-Safe Functions option.
Extensions beyond the ISO C standard are marked.
The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.
The DESCRIPTION of `asctime_r()` is updated to describe the format of the string returned.
The `restrict` keyword is added to the `asctime_r()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
asin, asinf, asinl — arc sine function

SYNOPSIS
#include <math.h>

double asin(double x);
float asinf(float x);
long double asinl(long double x);

DESCRIPTION
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

These functions shall compute the principal value of the arc sine of their argument \( x \). The value
of \( x \) should be in the range \([-1,1]\).

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

RETURN VALUE
Upon successful completion, these functions shall return the arc sine of \( x \), in the range
\([-\pi/2,\pi/2]\) radians.

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.

If \( x \) is NaN, a NaN shall be returned.

If \( x \) is ±0, \( x \) shall be returned.

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

If \( x \) is subnormal, a range error may occur and \( x \) should be returned.

ERRORS
These functions shall fail if:

Domain Error The \( x \) argument is finite and is not in the range \([-1,1]\), or is ±Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then \( 
\text{errno} \) 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.

These functions may fail if:

Range Error The value of \( x \) is subnormal.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then
\( \text{errno} \) 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.
EXAMPLES
None.

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

RATIONALE
None.

FUTURE DIRECTIONS
None.

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

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The asinf() and asinl() functions are added for alignment with the ISO/IEC 9899:1999 standard.

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

NAME

asinh, asinhf, asinhl — inverse hyperbolic sine functions

SYNOPSIS

#include <math.h>

double asinh(double x);

float asinhf(float x);

long double asinhl(long double x);

DESCRIPTION

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.

These functions shall compute the inverse hyperbolic sine of their argument \( x \).

An application wishing to check for error situations should set \( \text{errno} \) to zero and call \( \text{feclearexcept}(\text{FE_ALL_EXCEPT}) \) before calling these functions. On return, if \( \text{errno} \) is non-zero or \( \text{fetestexcept}(\text{FE_INVALID} \mid \text{FE_DIVBYZERO} \mid \text{FE_OVERFLOW} \mid \text{FE_UNDERFLOW}) \) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the inverse hyperbolic sine of their argument.

If \( x \) is \( \text{NaN} \), a \( \text{NaN} \) shall be returned.

If \( x \) is \( \pm 0 \), or \( \pm \infty \), \( x \) shall be returned.

If \( x \) is subnormal, a range error may occur and \( x \) should be returned.

ERRORS

These functions may fail if:

Range Error

The value of \( x \) is subnormal.

If the integer expression (\( \text{math_errhandling} \& \text{MATH_ERRNO} \)) is non-zero, then \( \text{errno} \) shall be set to \( \text{[ERANGE]} \). If the integer expression (\( \text{math_errhandling} \& \text{MATH_ERREXCEPT} \)) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

\( \text{feclearexcept}() \), \( \text{fetestexcept}() \), \( \text{sinh}() \), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, \(<\text{math.h}>\)
CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The \texttt{asinh()} function is no longer marked as an extension.

The \texttt{asinhf()} and \texttt{asinhl()} functions are added for alignment with the ISO/IEC 9899:1999 standard.

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

NAME
asinl — arc sine function

SYNOPSIS
#include <math.h>
long double asinl(long double x);

DESCRIPTION
Refer to asin().
NAME
assert — insert program diagnostics

SYNOPSIS
#include <assert.h>
void assert(scalar expression);

DESCRIPTION
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 assert() macro shall insert diagnostics into programs; it shall expand to a void expression. When it is executed, if expression (which shall have a scalar type) is false (that is, compares equal to 0), assert() shall write information about the particular call that failed on stderr and shall call abort().

The information written about the call that failed shall include the text of the argument, the name of the source file, the source file line number, and the name of the enclosing function; the latter are, respectively, the values of the preprocessing macros __FILE__ and __LINE__ and of the identifier __func__.

Forcing a definition of the name NDEBUG, either from the compiler command line or with the preprocessor control statement #define NDEBUG ahead of the #include <assert.h> statement, shall stop assertions from being compiled into the program.

RETURN VALUE
The assert() macro shall not return a value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
abort(), stderr, the Base Definitions volume of IEEE Std 1003.1-2001, <assert.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The prototype for the expression argument to assert() is changed from int to scalar for alignment with the ISO/IEC 9899: 1999 standard.

The DESCRIPTION of assert() is updated for alignment with the ISO/IEC 9899: 1999 standard.
NAME
atan, atanf, atanl — arc tangent function

SYNOPSIS
#include <math.h>

double atan(double x);
float atanf(float x);
long double atanl(long double x);

DESCRIPTION
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

These functions shall compute the principal value of the arc tangent of their argument x.
An application wishing to check for error situations should set errno to zero and call
fcallclexcept(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.

RETURN VALUE
Upon successful completion, these functions shall return the arc tangent of x in the range
[−π/2, π/2] radians.

If x is NaN, a NaN shall be returned.
If x is ±0, x shall be returned.
If x is ±Inf, ±π/2 shall be returned.
If x is subnormal, a range error may occur and x should be returned.

ERRORS
These functions may fail if:

Range Error The value of x is subnormal.

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.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
atan2(), feclearexcept(), fetestexcept(), isnan(), tan(), the Base Definitions volume of
IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
CHANGE HISTORY

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

Issue 5

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

Issue 6

The atanf() and atanl() functions are added for alignment with the ISO/IEC 9899:1999 standard.

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

NAME
atan2, atan2f, atan2l — arc tangent functions

SYNOPSIS
#include <math.h>
double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);

DESCRIPTION
The functionality described on this reference page is aligned with the ISO C standard. Any
collision between the requirements described here and the ISO C standard is unintentional. This
These functions shall compute the principal value of the arc tangent of
\( y/x \), using the signs of
both arguments to determine the quadrant of the return value.
An application wishing to check for error situations should set \texttt{errno} to zero and call
\texttt{fclearexcept}(FE_ALL_EXCEPT) before calling these functions. On return, if \texttt{errno} is non-zero or
\texttt{fetestexcept}(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE
Upon successful completion, these functions shall return the arc tangent of \( y/x \) in the range
\([-\pi, \pi]\) radians.
If \( y \) is \( \pm 0 \) and \( x \) is \( < 0 \), \( \pm \pi \) shall be returned.
If \( y \) is \( \pm 0 \) and \( x \) is \( > 0 \), \( \pm 0 \) shall be returned.
If \( y \) is \( < 0 \) and \( x \) is \( \pm 0 \), \( -\pi/2 \) shall be returned.
If \( y \) is \( > 0 \) and \( x \) is \( \pm 0 \), \( \pi/2 \) shall be returned.
If \( x \) is \( 0 \), a pole error shall not occur.
If either \( x \) or \( y \) is \( \text{NaN} \), a \( \text{NaN} \) shall be returned.
If the result underflows, a range error may occur and \( y/x \) should be returned.
If \( y \) is \( \pm 0 \) and \( x \) is \( -0 \), \( \pm \pi \) shall be returned.
If \( y \) is \( \pm 0 \) and \( x \) is \( +0 \), \( \pm 0 \) shall be returned.
For finite values of \( \pm y > 0 \), if \( x \) is \( -\text{Inf} \), \( \pm \pi \) shall be returned.
For finite values of \( \pm y > 0 \), if \( x \) is \( +\text{Inf} \), \( \pm 0 \) shall be returned.
For finite values of \( x \), if \( y \) is \( \pm\text{Inf} \), \( \pm \pi/2 \) shall be returned.
If \( y \) is \( \pm\text{Inf} \) and \( x \) is \( -\text{Inf} \), \( \pm 3\pi/4 \) shall be returned.
If \( y \) is \( \pm\text{Inf} \) and \( x \) is \( +\text{Inf} \), \( \pm \pi/4 \) shall be returned.
If both arguments are \( 0 \), a domain error shall not occur.

ERRORS
These functions may fail if:

Range Error                The result underflows.
If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
then \texttt{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.

**EXAMPLES**
None.

**APPLICATION USAGE**
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.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
atan(), 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>

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

**Issue 6**
The atan2f() and atan2l() functions are added for alignment with the ISO/IEC 9899:1999 standard.

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

NAME
atanf — arc tangent function

SYNOPSIS
#include <math.h>
float atanf(float x);

DESCRIPTION
Refer to atan().
NAME
atanh, atanhf, atanhl — inverse hyperbolic tangent functions

SYNOPSIS
#include <math.h>
double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);

DESCRIPTION
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.

These functions shall compute the inverse hyperbolic tangent of their argument x.

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.

RETURN VALUE
Upon successful completion, these functions shall return the inverse hyperbolic tangent of their argument.

If x is ±1, a pole error shall occur, and atanh(), atanhf(), and atanhl() shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively, with the same sign as the correct value of the function.

For finite |x| > 1, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If x is NaN, a NaN shall be returned.

If x is ±0, x shall be returned.

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

If x is subnormal, a range error may occur and x should be returned.

ERRORS
These functions shall fail if:

Domain Error The x argument is finite and not in the range [−1,1], or is ±Inf.

If the integer expression (math_errno & MATH_ERRNOM) is non-zero, then errno shall be set to [EDOM]. If the integer expression (math_errno & MATH_ERRNOREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

Pole Error The x argument is ±1.

If the integer expression (math_errno & MATH_ERRNOM) is non-zero, then errno shall be set to [ERANGE]. If the integer expression (math_errno & MATH_ERRNOREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.
These functions may fail if:

**Range Error**

The value of \( x \) is subnormal.

If the integer expression (math_errno & MATH_ERRNO) is non-zero, then \( errno \) shall be set to \[ERANGE\]. If the integer expression (math_errno & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

**EXAMPLES**

None.

**APPLICATION USAGE**

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

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

feclerexcept( ), fetestexcept( ), tanh( ), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, \(<math.h>\)

**CHANGE HISTORY**

First released in Issue 4, Version 2.

**Issue 5**

Moved from X/OPEN UNIX extension to BASE.

**Issue 6**

The \( \text{atanh}(\) function is no longer marked as an extension.

The \( \text{atanhf}(\) and \( \text{atanhl}(\) functions are added for alignment with the ISO/IEC 9899:1999 standard.

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

NAME
atanl — arc tangent function

SYNOPSIS
#include <math.h>
long double atanl(long double x);

DESCRIPTION
Refer to atan().
atexit() — register a function to run at process termination

#include <stdlib.h>

int atexit(void (*func)(void));

The atexit() function shall register the function pointed to by func, to be called without arguments at normal program termination. At normal program termination, all functions registered by the atexit() function shall be called, 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. Normal termination occurs either by a call to exit() or a return from main(). At least 32 functions can be registered with atexit().

CX After a successful call to any of the exec functions, any functions previously registered by atexit() shall no longer be registered.

RETURN VALUE
Upon successful completion, atexit() shall return 0; otherwise, it shall return a non-zero value.

ERRORS
No errors are defined.

APPLICATION USAGE
The functions registered by a call to atexit() must return to ensure that all registered functions are called.

The application should call sysconf() to obtain the value of {ATEXIT_MAX}, the number of functions that can be registered. There is no way for an application to tell how many functions have already been registered with atexit().

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
exit(), sysconf(), the Base Definitions volume of IEEE Std 1003.1-2001, stdlib.h

CHANGE HISTORY
First released in Issue 4. Derived from the ANSI C standard.

Issue 6
Extensions beyond the ISO C standard are marked.

The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
atof — convert a string to a double-precision number

SYNOPSIS
#include <stdlib.h>
double atof(const char *str);

DESCRIPTION
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 call atof(str) shall be equivalent to:
strtod(str, (char **)NULL),
except that the handling of errors may differ. If the value cannot be represented, the behavior is undefined.

RETURN VALUE
The atof() function shall return the converted value if the value can be represented.

ERRORS
No errors are defined.

APPLICATION USAGE
The atof() function is subsumed by strtod() but is retained because it is used extensively in existing code. If the number is not known to be in range, strtod() should be used because atof() is not required to perform any error checking.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strtod(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
atoi — convert a string to an integer

SYNOPSIS
#include <stdlib.h>

int atoi(const char *str);

DESCRIPTION
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 call atoi(str) shall be equivalent to:

(int) strtol(str, (char **)NULL, 10)

except that the handling of errors may differ. If the value cannot be represented, the behavior is undefined.

RETURN VALUE
The atoi() function shall return the converted value if the value can be represented.

ERRORS
No errors are defined.

EXAMPLES

Converting an Argument
The following example checks for proper usage of the program. If there is an argument and the decimal conversion of this argument (obtained using atoi()) is greater than 0, then the program has a valid number of minutes to wait for an event.

#include <stdlib.h>
#include <stdio.h>
...

int minutes_to_event;

... if (argc < 2 || ((minutes_to_event = atoi (argv[1]))) <= 0) {
    fprintf(stderr, "Usage: %s minutes\n", argv[0]); exit(1);
}

APPLICATION USAGE
The atoi() function is subsumed by strtol() but is retained because it is used extensively in existing code. If the number is not known to be in range, strtol() should be used because atoi() is not required to perform any error checking.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strtol(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>
CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.
atol()  

NAME
atol, atoll — convert a string to a long integer

SYNOPSIS
#include <stdlib.h>

long atol(const char *str);
long long atoll(const char *nptr);

DESCRIPTION
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 call atol(str) shall be equivalent to:
strtol(str, (char **)NULL, 10)
The call atoll(str) shall be equivalent to:
strtoll(nptr, (char **)NULL, 10)
except that the handling of errors may differ. If the value cannot be represented, the behavior is undefined.

RETURN VALUE
These functions shall return the converted value if the value can be represented.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
The atol() function is subsumed by strtol() but is retained because it is used extensively in existing code. If the number is not known to be in range, strtol() should be used because atol() is not required to perform any error checking.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strtol(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The atoll() function is added for alignment with the ISO/IEC 9899: 1999 standard.
NAME
basename — return the last component of a pathname

SYNOPSIS
XSI
#include <libgen.h>
char *basename(char *path);

DESCRIPTION
The basename() function shall take the pathname pointed to by path and return a pointer to the
final component of the pathname, deleting any trailing ‘/’ characters.

If the string consists entirely of the ‘/’ character, basename() shall return a pointer to the string
"/". If the string is exactly "///", it is implementation-defined whether ‘/’ or "///" is
returned.

If path is a null pointer or points to an empty string, basename() shall return a pointer to the
string ".".

The basename() function may modify the string pointed to by path, and may return a pointer to
static storage that may then be overwritten by a subsequent call to basename().

The basename() function need not be reentrant. A function that is not required to be reentrant is
not required to be thread-safe.

RETURN VALUE
The basename() function shall return a pointer to the final component of path.

ERRORS
No errors are defined.

EXAMPLES
Using basename()
The following program fragment returns a pointer to the value lib, which is the base name of
/usr/lib.
#include <libgen.h>
...
char *name = "/usr/lib";
char *base;
base = basename(name);
...

Sample Input and Output Strings for basename()
In the following table, the input string is the value pointed to by path, and the output string is
the return value of the basename() function.

<table>
<thead>
<tr>
<th>Input String</th>
<th>Output String</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;/usr/lib&quot;</td>
<td>&quot;lib&quot;</td>
</tr>
<tr>
<td>&quot;/usr/&quot;</td>
<td>&quot;usr&quot;</td>
</tr>
<tr>
<td>&quot;/&quot;</td>
<td>&quot;/&quot;</td>
</tr>
<tr>
<td>&quot;///&quot;</td>
<td>&quot;///&quot;</td>
</tr>
<tr>
<td>&quot;/usr///lib///&quot;</td>
<td>&quot;lib&quot;</td>
</tr>
</tbody>
</table>
See also `dirname()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<libgen.h>`, the Shell and Utilities volume of IEEE Std 1003.1-2001, `basename`.

Change History
--------

First released in Issue 4, Version 2.

Issue 5
--------

Moved from X/OPEN UNIX extension to BASE.

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

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6
--------

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
NAME
bcmp — memory operations (LEGACY)

SYNOPSIS
XSI
#include <strings.h>

int bcmp(const void *s1, const void *s2, size_t n);

DESCRIPTION
The bcmp() function shall compare the first n bytes of the area pointed to by s1 with the area pointed to by s2.

RETURN VALUE
The bcmp() function shall return 0 if s1 and s2 are identical; otherwise, it shall return non-zero.
Both areas are assumed to be n bytes long. If the value of n is 0, bcmp() shall return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
The memcmp() function is preferred over this function.
For maximum portability, it is recommended to replace the function call to bcmp() as follows:
#define bcmp(b1,b2,len) memcmp((b1), (b2), (size_t)(len))

RATIONALE
None.

FUTURE DIRECTIONS
This function may be withdrawn in a future version.

SEE ALSO
memcmp(), the Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>

CHANGE HISTORY
First released in Issue 4, Version 2.
Issue 5
Moved from X/OPEN UNIX extension to BASE.
Issue 6
This function is marked LEGACY.
NAME
bcopy — memory operations (LEGACY)

SYNOPSIS
XSI
#include <strings.h>
void bcopy(const void *s1, void *s2, size_t n);

DESCRIPTION
The bcopy() function shall copy n bytes from the area pointed to by s1 to the area pointed to by s2.
The bytes are copied correctly even if the area pointed to by s1 overlaps the area pointed to by s2.

RETURN VALUE
The bcopy() function shall not return a value.

ERRORS
No errors are defined.

APPLICATION USAGE
The memmove() function is preferred over this function.
The following are approximately equivalent (note the order of the arguments):
bcopy(s1, s2, n) ~= memmove(s2, s1, n)
For maximum portability, it is recommended to replace the function call to bcopy() as follows:
#define bcopy(bl, b2, len) (memmove((b2), (bl), (len)), (void) 0)

RATIONALE
None.

FUTURE DIRECTIONS
This function may be withdrawn in a future version.

SEE ALSO
memmove(), the Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
This function is marked LEGACY.
NAME
bind — bind a name to a socket

SYNOPSIS
#include <sys/socket.h>
int bind(int socket, const struct sockaddr *address,
        socklen_t address_len);

DESCRIPTION
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.

The bind() function takes the following arguments:

    socket Specifies the file descriptor of the socket to be bound.
    address 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.
    address_len Specifies the length of the sockaddr structure pointed to by the address
             argument.

The socket specified by socket may require the process to have appropriate privileges to use the
bind() function.

RETURN VALUE
Upon successful completion, bind() shall return 0; otherwise, −1 shall be returned and errno set
to indicate the error.

ERRORS
The bind() function shall fail if:

[EADDRINUSE] The specified address is already in use.
[EADDRNOTAVAIL]
The specified address is not available from the local machine.
[EAFNOSUPPORT]
The specified address is not a valid address for the address family of the
specified socket.
[EBADF] The socket argument is not a valid file descriptor.
[EINVAL] The socket is already bound to an address, and the protocol does not support
binding to a new address; or the socket has been shut down.
[ENOTSOCK] The socket argument does not refer to a socket.
[EOPNOTSUPP] The socket type of the specified socket does not support binding to an
address.

If the address family of the socket is AF_UNIX, then bind() shall fail if:

[EACCES] A component of the path prefix denies search permission, or the requested
name requires writing in a directory with a mode that denies write
permission.
[EINVAL] The address argument is a null pointer.
The `bind()` function may fail if:

- **[EACCES]** The specified address is protected and the current user does not have permission to bind to it.
- **[EINVAL]** The `address_len` argument is not a valid length for the address family.
- **[EISCONN]** The socket is already connected.
- **[ELOOP]** More than `SYMLOOP_MAX` symbolic links were encountered during resolution of the pathname in `address`.
- **[ENAMETOOLONG]** Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.
- **[ENOBUFS]** Insufficient resources were available to complete the call.

**EXAMPLES**

None.

**APPLICATION USAGE**

An application program can retrieve the assigned socket name with the `getsockname()` function.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`connect()`, `getsockname()`, `listen()`, `socket()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/socket.h>`

**CHANGE HISTORY**

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
bsd_signal — simplified signal facilities

SYNOPSIS
#include <signal.h>

void (*bsd_signal(int sig, void (*func)(int)))(int);

DESCRIPTION
The bsd_signal() function provides a partially compatible interface for programs written to historical system interfaces (see APPLICATION USAGE).

The function call bsd_signal(sig, func) shall be equivalent to the following:

```c
void (*bsd_signal(int sig, void (*func)(int)))(int)
{
  struct sigaction act, oact;
  act.sa_handler = func;
  act.sa_flags = SA_RESTART;
  sigemptyset(&act.sa_mask);
  sigaddset(&act.sa_mask, sig);
  if (sigaction(sig, &act, &oact) == -1)
    return(SIG_ERR);
  return(oact.sa_handler);
}
```

The handler function should be declared:

```c
void handler(int sig);
```

where sig is the signal number. The behavior is undefined if func is a function that takes more than one argument, or an argument of a different type.

RETURN VALUE
Upon successful completion, bsd_signal() shall return the previous action for sig. Otherwise, SIG_ERR shall be returned and errno shall be set to indicate the error.

ERRORS
Refer to sigaction().

EXAMPLES
None.

APPLICATION USAGE
This function is a direct replacement for the BSD signal() function for simple applications that are installing a single-argument signal handler function. If a BSD signal handler function is being installed that expects more than one argument, the application has to be modified to use sigaction(). The bsd_signal() function differs from signal() in that the SA_RESTART flag is set and the SA_RESETHAND is clear when bsd_signal() is used. The state of these flags is not specified for signal().

It is recommended that new applications use the sigaction() function.

RATIONALE
None.
FUTURE DIRECTIONS
None.

SEE ALSO
sigaction(), sigaddset(), sigemptyset(), signal(), the Base Definitions volume of
IEEE Std 1003.1-2001, <signal.h>

CHANGE HISTORY
First released in Issue 4, Version 2.
Issue 5
Moved from X/OPEN UNIX extension to BASE.
Issue 6
This function is marked obsolescent.
NAME
bsearch — binary search a sorted table

SYNOPSIS
#include <stdlib.h>

void *bsearch(const void *key, const void *base, size_t nel,
               size_t width, int (*compar)(const void *, const void *));

DESCRIPTION
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

The bsearch() function shall search an array of nel objects, the initial element of which is pointed
to by base, for an element that matches the object pointed to by key. The size of each element in
the array is specified by width.

The comparison function pointed to by compar shall be called with two arguments that point to
the key object and to an array element, in that order.

The application shall ensure that the function returns an integer less than, equal to, or greater
than 0 if the key object is considered, respectively, to be less than, to match, or to be greater than
the array element. The application shall ensure that the array consists of all the elements that
compare less than, all the elements that compare equal to, and all the elements that compare
greater than the key object, in that order.

RETURN VALUE
The bsearch() function shall return a pointer to a matching member of the array, or a null pointer
if no match is found. If two or more members compare equal, which member is returned is
unspecified.

EXAMPLES
The example below searches a table containing pointers to nodes consisting of a string and its
length. The table is ordered alphabetically on the string in the node pointed to by each entry.

The code fragment below reads in strings and either finds the corresponding node and prints out
the string and its length, or prints an error message.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TABSIZE 1000

struct node {
    char *string;
    int length;
};

struct node table[TABSIZE];

struct node *node_ptr, node;

/* Routine to compare 2 nodes. */
```c
int node_compare(const void *, const void *);
char str_space[20]; /* Space to read string into. */
.
.
.node.string = str_space;
while (scanf("%s", node.string) != EOF) {
    node_ptr = (struct node *)bsearch((void *)&node,
        (void *)table, TABSIZE,
        sizeof(struct node), node_compare);
    if (node_ptr != NULL) {
        (void)printf("string = %20s, length = %d\n",
            node_ptr->string, node_ptr->length);
    } else {
        (void)printf("not found: %s\n", node.string);
    }
}

/*
   This routine compares two nodes based on an
   alphabetical ordering of the string field.
*/

int node_compare(const void *node1, const void *node2)
{
    return strcoll(((const struct node *)node1)->string,
        ((const struct node *)node2)->string);
}

APPLICATION USAGE

The pointers to the key and the element at the base of the table should be of type pointer-to-
element.

The comparison function need not compare every byte, so arbitrary data may be contained in
the elements in addition to the values being compared.

In practice, the array is usually sorted according to the comparison function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

hcreate(), lsearch(), qsort(), tsearch(), the Base Definitions volume of IEEE Std 1003.1-2001,
<stdlib.h>

CHANGE HISTORY

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

Issue 6

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
```
NAME
btowc — single byte to wide character conversion

SYNOPSIS
#include <stdio.h>
#include <wchar.h>

wint_t btowc(int c);

DESCRIPTION
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 btowc() function shall determine whether c constitutes a valid (one-byte) character in the initial shift state.

The behavior of this function shall be affected by the LC_CTYPE category of the current locale.

RETURN VALUE
The btowc() function shall return WEOF if c has the value EOF or if (unsigned char) c does not constitute a valid (one-byte) character in the initial shift state. Otherwise, it shall return the wide-character representation of that character.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wctob(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
NAME
bzero — memory operations (LEGACY)

SYNOPSIS
#include <strings.h>

void bzero(void *s, size_t n);

DESCRIPTION
The bzero() function shall place n zero-valued bytes in the area pointed to by s.

RETURN VALUE
The bzero() function shall not return a value.

ERRORS
No errors are defined.

APPLICATION USAGE
The memset() function is preferred over this function.
For maximum portability, it is recommended to replace the function call to bzero() as follows:
#define bzero(b,len) (memset((b), ’\0’, (len)), (void) 0)

RATIONALE
None.

FUTURE DIRECTIONS
This function may be withdrawn in a future version.

SEE ALSO
memset(), the Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>

CHANGE HISTORY
First released in Issue 4, Version 2.
Issue 5
Moved from X/OPEN UNIX extension to BASE.
Issue 6
This function is marked LEGACY.
NAME

cabs, cabsf, cabsl — return a complex absolute value

SYNOPSIS

#include <complex.h>

double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);

DESCRIPTION

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.

These functions shall compute the complex absolute value (also called norm, modulus, or magnitude) of z.

RETURN VALUE

These functions shall return the complex absolute value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY

NAME

cacos, cacosf, cacosl — complex arc cosine functions

SYNOPSIS

#include <complex.h>

double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);

DESCRIPTION

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.

These functions shall compute the complex arc cosine of z, with branch cuts outside the interval \([-1, +1]\) along the real axis.

RETURN VALUE

These functions shall return the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval \([0, \pi]\) along the real axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cacos(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY

NAME
\text{cacosh, cacoshf, cacoshl} — complex arc hyperbolic cosine functions

SYNOPSIS
\begin{verbatim}
#include <complex.h>

double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
\end{verbatim}

DESCRIPTION
\text{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.}
\text{These functions shall compute the complex arc hyperbolic cosine of } z, \text{ with a branch cut at values less than 1 along the real axis.}

RETURN VALUE
\text{These functions shall return the complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis and in the interval } [-i\pi, +i\pi] \text{ along the imaginary axis.}

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
\text{ccosh()}, the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY
NAME

cacosl — complex arc cosine functions

SYNOPSIS

#include <complex.h>

long double complex cacosl(long double complex z);

DESCRIPTION

Refer to cacos().
NAME
calloc — a memory allocator

SYNOPSIS
#include <stdlib.h>
void *calloc(size_t nelem, size_t elsize);

DESCRIPTION
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 calloc() function shall allocate unused space for an array of nelem elements each of whose size in bytes is elsize. The space shall be initialized to all bits 0.

The order and contiguity of storage allocated by successive calls to calloc() 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 or an array of such objects 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 shall point 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.

RETURN VALUE
Upon successful completion with both nelem and elsize non-zero, calloc() shall return a pointer to the allocated space. If either nelem or elsize is 0, then either a null pointer or a unique pointer value that can be successfully passed to free() shall be returned. Otherwise, it shall return a null pointer and set errno to indicate the error.

ERRORS
The calloc() function shall fail if:

[ENOMEM] Insufficient memory is available.

EXAMPLES
None.

APPLICATION USAGE
There is now no requirement for the implementation to support the inclusion of <malloc.h>.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
free(), malloc(), realloc(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The setting of *errno* and the [ENOMEM] error condition are mandatory if an insufficient memory condition occurs.
NAME
carg, cargf, cargl — complex argument functions

SYNOPSIS
#include <complex.h>

double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);

DESCRIPTION
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

These functions shall compute the argument (also called phase angle) of z, with a branch cut
along the negative real axis.

RETURN VALUE
These functions shall return the value of the argument in the interval \([-\pi, +\pi]\).

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
cimag(), conj(), cproj(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY
NAME
casin, casinf, casinl — complex arc sine functions

SYNOPSIS
#include <complex.h>

double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);

DESCRIPTION
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

These functions shall compute the complex arc sine of z, with branch cuts outside the interval
[−1, +1] along the real axis.

RETURN VALUE
These functions shall return the complex arc sine value, in the range of a strip mathematically
unbounded along the imaginary axis and in the interval [−π/2, +π/2] along the real axis.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
csin( ), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY
NAME

casinh, casinhf, casinhl — complex arc hyperbolic sine functions

SYNOPSIS

#include <complex.h>

double complex casinh(double complex z);
floating complex casinhf(floating complex z);
long double complex casinhl(long double complex z);

DESCRIPTION

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

These functions shall compute the complex arc hyperbolic sine of \(z\), with branch cuts outside the
interval \([-i, +i]\) along the imaginary axis.

RETURN VALUE

These functions shall return the complex arc hyperbolic sine value, in the range of a strip
mathematically unbounded along the real axis and in the interval \([-i\pi/2, +i\pi/2]\) along the
imaginary axis.

ERRORS

No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO

csinh(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY

casinl() — complex arc sine functions

#include <complex.h>

long double complex casinl(long double complex z);

Refer to casin().
NAME

catan, catanf, catanl — complex arc tangent functions

SYNOPSIS

#include <complex.h>

double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);

DESCRIPTION

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.

These functions shall compute the complex arc tangent of \( z \), with branch cuts outside the interval \([-i, +i]\) along the imaginary axis.

RETURN VALUE

These functions shall return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval \([-\pi/2, +\pi/2]\) along the real axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ctan(), the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<complex.h>}

CHANGE HISTORY

NAME
catanh, catanhf, catanhl — complex arc hyperbolic tangent functions

SYNOPSIS
#include <complex.h>

double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);

DESCRIPTION
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.

These functions shall compute the complex arc hyperbolic tangent of \( z \), with branch cuts outside the interval \([-1, +1]\) along the real axis.

RETURN VALUE
These functions shall return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval \([-i\pi/2, +i\pi/2]\) along the imaginary axis.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
ctan(), the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<complex.h>}

CHANGE HISTORY
NAME
catanl — complex arc tangent functions

SYNOPSIS
#include <complex.h>
long double complex catanl(long double complex z);

DESCRIPTION
Refer to catan().
NAME
catclose — close a message catalog descriptor

SYNOPSIS

```
#include <nl_types.h>

int catclose(nl_catd catd);
```

DESCRIPTION

The `catclose()` function shall close the message catalog identified by `catd`. If a file descriptor is used to implement the type `nl_catd`, that file descriptor shall be closed.

RETURN VALUE

Upon successful completion, `catclose()` shall return 0; otherwise, −1 shall be returned, and `errno` set to indicate the error.

ERRORS

The `catclose()` function may fail if:

- [EBADF] The catalog descriptor is not valid.
- [EINTR] The `catclose()` function was interrupted by a signal.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`catgets()`, `catopen()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<nl_types.h>`

CHANGE HISTORY

First released in Issue 2.
NAME

catgets — read a program message

SYNOPSIS

XSI
#include <nl_types.h>

char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);

DESCRIPTION

The catgets() function shall attempt to read message msg_id, in set set_id, from the message catalog identified by catd. The catd argument is a message catalog descriptor returned from an earlier call to catopen(). The s argument points to a default message string which shall be returned by catgets() if it cannot retrieve the identified message.

The catgets() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

If the identified message is retrieved successfully, catgets() shall return a pointer to an internal buffer area containing the null-terminated message string. If the call is unsuccessful for any reason, s shall be returned and errno may be set to indicate the error.

ERRORS

The catgets() function may fail if:

[EBADF] The catd argument is not a valid message catalog descriptor open for reading.
[EBADMSG] The message identified by set_id and msg_id in the specified message catalog did not satisfy implementation-defined security criteria.
[EINTR] The read operation was terminated due to the receipt of a signal, and no data was transferred.
[EINVAL] The message catalog identified by catd is corrupted.
[ENOMSG] The message identified by set_id and msg_id is not in the message catalog.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catclose(), catopen(), the Base Definitions volume of IEEE Std 1003.1-2001, <nl_types.h>

CHANGE HISTORY

First released in Issue 2.

Issue 5

A note indicating that this function need not be reentrant is added to the DESCRIPTION.
In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
NAME

catopen — open a message catalog

SYNOPSIS

```
#include <nl_types.h>

nl_catd catopen(const char *name, int oflag);
```

DESCRIPTION

The `catopen()` function shall open a message catalog and return a message catalog descriptor. The `name` argument specifies the name of the message catalog to be opened. If `name` contains a `'/',` then `name` specifies a complete name for the message catalog. Otherwise, the environment variable `NLSPATH` is used with `name` substituted for the `%N` conversion specification (see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables). If `NLSPATH` exists in the environment when the process starts, then if the process has appropriate privileges, the behavior of `catopen()` is undefined. If `NLSPATH` does not exist in the environment, or if a message catalog cannot be found in any of the components specified by `NLSPATH`, then an implementation-defined default path shall be used. This default may be affected by the setting of `LC_MESSAGES` if the value of `oflag` is `NL_CAT_LOCALE`, or the `LANG` environment variable if `oflag` is `0`.

A message catalog descriptor shall remain valid in a process until that process closes it, or a successful call to one of the `exec` functions. A change in the setting of the `LC_MESSAGES` category may invalidate existing open catalogs.

If a file descriptor is used to implement message catalog descriptors, the `FD_CLOEXEC` flag shall be set; see `<fcntl.h>`.

If the value of the `oflag` argument is `0`, the `LANG` environment variable is used to locate the catalog without regard to the `LC_MESSAGES` category. If the `oflag` argument is `NL_CAT_LOCALE`, the `LC_MESSAGES` category is used to locate the message catalog (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 8.2, Internationalization Variables).

RETURN VALUE

Upon successful completion, `catopen()` shall return a message catalog descriptor for use on subsequent calls to `catgets()` and `catclose()`. Otherwise, `catopen()` shall return `(-1)` and set `errno` to indicate the error.

ERRORS

The `catopen()` function may fail if:

- **[EACCES]** Search permission is denied for the component of the path prefix of the message catalog or read permission is denied for the message catalog.
- **[EMFILE]** `[OPEN_MAX]` file descriptors are currently open in the calling process.
- **[ENAMETOOLONG]** The length of a pathname of the message catalog exceeds `[PATH_MAX]` or a pathname component is longer than `[NAME_MAX]`.
- **[ENAMETOOLONG]** Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `[PATH_MAX]`.
- **[ENFILE]** Too many files are currently open in the system.
- **[ENOENT]** The message catalog does not exist or the `name` argument points to an empty string.
catopen()

System Interfaces

5807 [ENOMEM] Insufficient storage space is available.
5808 [ENOTDIR] A component of the path prefix of the message catalog is not a directory.

EXAMPLES
None.

APPLICATION USAGE
Some implementations of catopen() use malloc() to allocate space for internal buffer areas. The catopen() function may fail if there is insufficient storage space available to accommodate these buffers.

Conforming applications must assume that message catalog descriptors are not valid after a call to one of the exec functions.

Application writers should be aware that guidelines for the location of message catalogs have not yet been developed. Therefore they should take care to avoid conflicting with catalogs used by other applications and the standard utilities.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
catclose(), catgets(), the Base Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>, <nl_types.h>, the Shell and Utilities volume of IEEE Std 1003.1-2001

CHANGE HISTORY
First released in Issue 2.
NAME
cbrt, cbrtf, cbrtl — cube root functions

SYNOPSIS
#include <math.h>

double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);

DESCRIPTION
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.

These functions shall compute the real cube root of their argument x.

RETURN VALUE
Upon successful completion, these functions shall return the cube root of x.

If x is NaN, a NaN shall be returned.
If x is ±0 or ±Inf, x shall be returned.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
For some applications, a true cube root function, which returns negative results for negative arguments, is more appropriate than pow(x, 1.0/3.0), which returns a NaN for x less than 0.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
The cbrt() function is no longer marked as an extension.
The cbrtf() and cbrtl() functions are added for alignment with the ISO/IEC 9899: 1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899: 1999 standard.
NAME
ccos, ccosf, ccosl — complex cosine functions

SYNOPSIS
#include <complex.h>

double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);

DESCRIPTION
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

These functions shall compute the complex cosine of z.

RETURN VALUE
These functions shall return the complex cosine value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
cacos(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY
**NAME**
ccosh, ccoshf, ccoshl — complex hyperbolic cosine functions

**SYNOPSIS**
#include <complex.h>

double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);

**DESCRIPTION**
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.

These functions shall compute the complex hyperbolic cosine of z.

**RETURN VALUE**
These functions shall return the complex hyperbolic cosine value.

**ERRORS**
No errors are defined.

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
cacosh(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

**CHANGE HISTORY**
NAME
ccosl — complex cosine functions

SYNOPSIS
#include <complex.h>

long double complex ccosl(long double complex z);

DESCRIPTION
Refer to ccos().
NAME
ceil, ceilf, ceill — ceiling value function

SYNOPSIS
#include <math.h>
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);

DESCRIPTION
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

These functions shall compute the smallest integral value not less than x.

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.

RETURN VALUE
Upon successful completion, ceil(), ceilf(), and ceill() shall return the smallest integral value not
less than x, expressed as a type double, float, or long double, respectively.

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

If x is ±0 or ±Inf, x shall be returned.

XSI If the correct value would cause overflow, a range error shall occur and ceil(), ceilf(), and ceill()
shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

ERRORS
These functions shall fail if:

XSI Range Error The result overflows.

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.

EXAMPLES
None.

APPLICATION USAGE
The integral value returned by these functions need 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.

The ceil() function can only overflow when the floating-point representation has
DBL_MANT_DIG > DBL_MAX_EXP.

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.
ceil()

RATIONALE
None.

FUTURE DIRECTIONS
None.

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

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The ceilf() and ceill() functions are added for alignment with the ISO/IEC 9899:1999 standard.

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

NAME
cexp, cexpf, cexpl — complex exponential functions

SYNOPSIS
#include <complex.h>

double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);

DESCRIPTION
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

These functions shall compute the complex exponent of z, defined as e^z.

RETURN VALUE
These functions shall return the complex exponential value of z.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
clog( ), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY
NAME

cfgetispeed — get input baud rate

SYNOPSIS

#include <termios.h>

speed_t cfgetispeed(const struct termios *termios_p);

DESCRIPTION

The cfgetispeed() function shall extract the input baud rate from the termios structure to which
the termios_p argument points.

This function shall return exactly the value in the termios data structure, without interpretation.

RETURN VALUE

Upon successful completion, cfgetispeed() shall return a value of type speed_t representing the
input baud rate.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The term “baud” is used historically here, but is not technically correct. This is properly “bits
per second”, which may not be the same as baud. However, the term is used because of the
historical usage and understanding.

The cfgetospeed(), cfgetispeed(), cfsetospeed(), and cfsetispeed() functions do not take arguments as
numbers, but rather as symbolic names. There are two reasons for this:

1. Historically, numbers were not used because of the way the rate was stored in the data
structure. This is retained even though a function is now used.

2. More importantly, only a limited set of possible rates is at all portable, and this constrains
the application to that set.

There is nothing to prevent an implementation accepting an extension a number (such as 126),
and since the encoding of the Bxxx symbols is not specified, this can be done to avoid
introducing ambiguity.

Setting the input baud rate to zero was a mechanism to allow for split baud rates. Clarifications
in this volume of IEEE Std 1003.1-2001 have made it possible to determine whether split rates are
supported and to support them without having to treat zero as a special case. Since this
functionality is also confusing, it has been declared obsolescent. The 0 argument referred to is
the literal constant 0, not the symbolic constant B0. This volume of IEEE Std 1003.1-2001 does not
preclude B0 from being defined as the value 0; in fact, implementations would likely benefit
from the two being equivalent. This volume of IEEE Std 1003.1-2001 does not fully specify
whether the previous cfsetispeed() value is retained after a tcgetattr() as the actual value or as
zero. Therefore, conforming applications should always set both the input speed and output
speed when setting either.

In historical implementations, the baud rate information is traditionally kept in c_cflag.
Applications should be written to presume that this might be the case (and thus not blindly copy
c_cflag), but not to rely on it in case it is in some other field of the structure. Setting the c_cflag
field absolutely after setting a baud rate is a non-portable action because of this. In general, the
unused parts of the flag fields might be used by the implementation and should not be blindly

copied from the descriptions of one terminal device to another.

FUTURE DIRECTIONS

None.

SEE ALSO
cfgetospeed(), cfsetispeed(), cfsetospeed(), tcgetattr(), the Base Definitions volume of
IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.
NAME
cfgetospeed — get output baud rate

SYNOPSIS
#include <termios.h>
speed_t cfgetospeed(const struct termios *termios_p);

DESCRIPTION
The cfgetospeed() function shall extract the output baud rate from the termios structure to which
the termios_p argument points.
This function shall return exactly the value in the termios data structure, without interpretation.

RETURN VALUE
Upon successful completion, cfgetospeed() shall return a value of type speed_t representing the
output baud rate.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
Refer to cfgetispeed().

FUTURE DIRECTIONS
None.

SEE ALSO
cfgetispeed(), cfsetispeed(), cfsetospeed(), tcgetattr(), the Base Definitions volume of
IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>

CHANGE HISTORY
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.
NAME
cfsetispeed — set input baud rate

SYNOPSIS
#include <termios.h>

int cfsetispeed(struct termios *termios_p, speed_t speed);

DESCRIPTION
The cfsetispeed() function shall set the input baud rate stored in the structure pointed to by
termios_p to speed.

There shall be no effect on the baud rates set in the hardware until a subsequent successful call
to tcsetattr() with the same termios structure. Similarly, errors resulting from attempts to set
baud rates not supported by the terminal device need not be detected until the tcsetattr() function is called.

RETURN VALUE
Upon successful completion, cfsetispeed() shall return 0; otherwise, −1 shall be returned, and
errno may be set to indicate the error.

ERRORS
The cfsetispeed() function may fail if:

[EINVAL] The speed value is not a valid baud rate.

[EINVAL] The value of speed is outside the range of possible speed values as specified in
<termios.h>.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
Refer to cfgetispeed().

FUTURE DIRECTIONS
None.

SEE ALSO
cfgetispeed(), cfgetospeed(), cfsetospeed(), tcsetattr(), the Base Definitions volume of
IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>

CHANGE HISTORY
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 6
The following new requirements on POSIX implementations derive from alignment with the
Single UNIX Specification:

• The optional setting of errno and the [EINVAL] error conditions are added.
NAME

cfsetospeed — set output baud rate

SYNOPSIS

#include <termios.h>

int cfsetospeed(struct termios *termios_p, speed_t speed);

DESCRIPTION

The cfsetospeed() function shall set the output baud rate stored in the structure pointed to by termios_p to speed.

There shall be no effect on the baud rates set in the hardware until a subsequent successful call to tcsetattr() with the same termios structure. Similarly, errors resulting from attempts to set baud rates not supported by the terminal device need not be detected until the tcsetattr() function is called.

RETURN VALUE

Upon successful completion, cfsetospeed() shall return 0; otherwise, it shall return −1 and errno may be set to indicate the error.

ERRORS

The cfsetospeed() function may fail if:

• [EINVAL] The speed value is not a valid baud rate.
• [EINVAL] The value of speed is outside the range of possible speed values as specified in <termios.h>.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to cfgetispeed().

FUTURE DIRECTIONS

None.

SEE ALSO

cfgetispeed(), cfgetospeed(), cfsetispeed(), tcsetattr(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 6

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

• The optional setting of errno and the [EINVAL] error conditions are added.
NAME
chdir — change working directory

SYNOPSIS
#include <unistd.h>
int chdir(const char *path);

DESCRIPTION
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 ‘/’.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned, the current working directory shall remain unchanged, and errno shall be set to indicate the error.

ERRORS
The chdir() function shall fail if:

[EACCES] Search permission is denied for any component of the pathname.

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

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

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

[ENOTDIR] A component of the pathname is not a directory.

The chdir() function may fail if:

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

[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}.

EXAMPLES
Changing the Current Working Directory
The following example makes the value pointed to by directory, /tmp, the current working directory.

#include <unistd.h>

char *directory = "/tmp";
int ret;
ret = chdir(directory);
APPLICATION USAGE
None.

RATIONALE
The chdir() function only affects the working directory of the current process.

FUTURE DIRECTIONS
None.

SEE ALSO
getcwd(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The APPLICATION USAGE section is added.
The following new requirements on POSIX implementations derive from alignment with the
Single UNIX Specification:
• The [ELOOP] mandatory error condition is added.
• A second [ENAMETOOLONG] is added as an optional error condition.
The following changes were made to align with the IEEE P1003.1a draft standard:
• The [ELOOP] optional error condition is added.
NAME
chmod — change mode of a file

SYNOPSIS
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);

DESCRIPTION
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.

S_ISUID, S_ISGID, S_ISVTX, and the file permission bits are described in <sys/stat.h>.

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().

Additional implementation-defined restrictions may cause the S_ISUID and S_ISGID bits in mode to be ignored.

The effect on file descriptors for files open at the time of a call to chmod() is implementation-defined.

Upon successful completion, chmod() shall mark for update the st_ctime field of the file.

RETURN VALUE
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.

ERRORS
The chmod() function shall fail if:

[EACCES] Search permission is denied on a component of the path prefix.

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

[ENAMETOOLONG] The length of the path argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

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

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

[EPERM] The effective user ID does not match the owner of the file and the process does not have appropriate privileges.

[EROFS] The named file resides on a read-only file system.

The chmod() function may fail if:

[EINTR] A signal was caught during execution of the function.

[EINVAL] The value of the mode argument is invalid.

[ELOOP] More than SYMLOOP_MAX symbolic links were encountered during resolution of the path argument.
[ENAMETOOLONG]

As a result of encountering a symbolic link in resolution of the path argument, the length of the substituted pathname strings exceeded {PATH_MAX}.

EXAMPLES

Setting Read Permissions for User, Group, and Others

The following example sets read permissions for the owner, group, and others.

```c
#include <sys/stat.h>
const char *path;
...
chmod(path, S_IRUSR|S_IRGRP|S_IROTH);
```

Setting Read, Write, and Execute Permissions for the Owner Only

The following example sets read, write, and execute permissions for the owner, and no permissions for group and others.

```c
#include <sys/stat.h>
const char *path;
...
chmod(path, S_IRWXU);
```

Setting Different Permissions for Owner, Group, and Other

The following example sets owner permissions for CHANGEFILE to read, write, and execute, group permissions to read and execute, and other permissions to read.

```c
#include <sys/stat.h>
#define CHANGEFILE "/etc/myfile"
...
chmod(CHANGEFILE, S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH);
```

Setting and Checking File Permissions

The following example sets the file permission bits for a file named /home/cnd/mod1, then calls the stat() function to verify the permissions.

```c
#include <sys/types.h>
#include <sys/stat.h>
int status;
struct stat buffer
...
chmod("/home/cnd/mod1", S_IRWXU|S_IRWXG|S_IROTH|S_IWOTH);
status = stat("/home/cnd/mod1", &buffer);
```

APPLICATION USAGE

In order to ensure that the S_ISUID and S_ISGID bits are set, an application requiring this should use stat() after a successful chmod() to verify this.

Any file descriptors currently open by any process on the file could possibly become invalid if the mode of the file is changed to a value which would deny access to that process. One
situation where this could occur is on a stateless file system. This behavior will not occur in a
conforming environment.

RATIONALE
This volume of IEEE Std 1003.1-2001 specifies that the S_ISGID bit is cleared by chmod() on a
regular file under certain conditions. This is specified on the assumption that regular files may
be executed, and the system should prevent users from making executable setgid() files perform
with privileges that the caller does not have. On implementations that support execution of
other file types, the S_ISGID bit should be cleared for those file types under the same
circumstances.

Implementations that use the S_ISUID bit to indicate some other function (for example,
mandatory record locking) on non-executable files need not clear this bit on writing. They
should clear the bit for executable files and any other cases where the bit grants special powers
to processes that change the file contents. Similar comments apply to the S_ISGID bit.

FUTURE DIRECTIONS
None.

SEE ALSO
chown(), mkdir(), mkfifo(), open(), stat(), statvfs(), the Base Definitions volume of
IEEE Std 1003.1-2001, <sys/stat.h>, <sys/types.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
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.

• The [EINVAL] and [EINTR] optional error conditions are added.

• A second [ENAMETOOLONG] is added as an optional error condition.

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

• The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
chown — change owner and group of a file

SYNOPSIS
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);

DESCRIPTION
The chown() function shall change the user and group ownership of a file.

The path argument points to a pathname naming a file. The user ID and group ID of the named
file shall be set to the numeric values contained in owner and group, respectively.

Only processes with an effective user ID equal to the user ID of the file or with appropriate
privileges may change the ownership of a file. If _POSIX_CHOWN_RESTRICTED is in effect for
path:

• Changing the user ID is restricted to processes with appropriate privileges.
• Changing the group ID is permitted to a process with an effective user ID equal to the user
  ID of the file, but without appropriate privileges, if and only if owner is equal to the file’s user
  ID or (uid_t)−1 and group is equal either to the calling process’ effective group ID or to one of
  its supplementary group IDs.

If the specified file is a regular file, one or more of the S_IXUSR, S_IXGRP, or S_IXOTH bits of
the file mode are set, and the process does not have appropriate privileges, the set-user-ID
(S_ISUID) and set-group-ID (S_ISGID) bits of the file mode shall be cleared upon successful
return from chown(). If the specified file is a regular file, one or more of the S_IXUSR, S_IXGRP,
or S_IXOTH bits of the file mode are set, and the process has appropriate privileges, it is
implementation-defined whether the set-user-ID and set-group-ID bits are altered. If the chown() function is successfully invoked on a file that is not a regular file and one or more of the
S_IXUSR, S_IXGRP, or S_IXOTH bits of the file mode are set, the set-user-ID and set-group-ID
bits may be cleared.

If owner or group is specified as (uid_t)−1 or (gid_t)−1, respectively, the corresponding ID of the
file shall not be changed. If both owner and group are −1, the times need not be updated.

Upon successful completion, chown() shall mark for update the st_ctime field of the file.

RETURN VALUE
Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned and errno set to
indicate the error. If −1 is returned, no changes are made in the user ID and group ID of the file.

ERRORS
The chown() function shall fail if:

[EACCES] Search permission is denied on a component of the path prefix.

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

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

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

[ENOENT] A component of path does not name an existing file or path is an empty string.
The chown() function may fail if:

- **[EIO]** An I/O error occurred while reading or writing to the file system.
- **[EINTR]** The chown() function was interrupted by a signal which was caught.
- **[EINVAL]** The owner or group ID supplied is not a value supported by the implementation.
- **[ELOOP]** More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the path argument.
- **[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].

**EXAMPLES**

None.

**APPLICATION USAGE**

Although chown() can be used on some implementations by the file owner to change the owner and group to any desired values, the only portable use of this function is to change the group of a file to the effective GID of the calling process or to a member of its group set.

**RATIONALE**

System III and System V allow a user to give away files; that is, the owner of a file may change its user ID to anything. This is a serious problem for implementations that are intended to meet government security regulations. Version 7 and 4.3 BSD permit only the superuser to change the user ID of a file. Some government agencies (usually not ones concerned directly with security) find this limitation too confining. This volume of IEEE Std 1003.1-2001 uses may to permit secure implementations while not disallowing System V.

System III and System V allow the owner of a file to change the group ID to anything. Version 7 permits only the superuser to change the group ID of a file. 4.3 BSD permits the owner to change the group ID of a file to its effective group ID or to any of the groups in the list of supplementary group IDs, but to no others.

The POSIX.1-1990 standard requires that the chown() function invoked by a non-appropriate privileged process clear the S_ISGID and the S_ISUID bits for regular files, and permits them to be cleared for other types of files. This is so that changes in accessibility do not accidentally cause files to become security holes. Unfortunately, requiring these bits to be cleared on non-executable data files also clears the mandatory file locking bit (shared with S_ISUID), which is an extension on many implementations (it first appeared in System V). These bits should only be required to be cleared on regular files that have one or more of their execute bits set.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

chmod(), pathconf(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>
CHANGE HISTORY

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

**Issue 6**

The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- The wording describing the optional dependency on `_POSIX_CHOWN_RESTRICTED` is restored.
- The `[EPERM]` error is restored as an error dependent on `_POSIX_CHOWN_RESTRICTED`. This is since its operand is a pathname and applications should be aware that the error may not occur for that pathname if the file system does not support `_POSIX_CHOWN_RESTRICTED`.

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.
- The value for `owner` of `(uid_t)−1` allows the use of −1 by the owner of a file to change the group ID only. A corresponding change is made for `group`.
- The `[ELOOP]` mandatory error condition is added.
- The `[EIO]` and `[EINTR]` optional error conditions are added.
- A second `[ENAMETOOLONG]` is added as an optional error condition.

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

- Clarification is added that the `S_ISUID` and `S_ISGID` bits do not need to be cleared when the process has appropriate privileges.
- The `[ELOOP]` optional error condition is added.
NAME

cimag, cimagf, cimagl — complex imaginary functions

SYNOPSIS

#include <complex.h>

double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);

DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any
contlict between the requirements described here and the ISO C standard is unintentional. This

These functions shall compute the imaginary part of z.

RETURN VALUE

These functions shall return the imaginary part value (as a real).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

For a variable z of complex type:

z == creal(z) + cimag(z)*I

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

carg(), conj(), cproj(), creal(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY

clearerr

NAME
clearerr — clear indicators on a stream

SYNOPSIS
#include <stdio.h>

void clearerr(FILE *stream);

DESCRIPTION
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
The clearerr() function shall clear the end-of-file and error indicators for the stream to which
stream points.

RETURN VALUE
The clearerr() function shall not return a value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
   clock — report CPU time used

SYNOPSIS
   #include <time.h>
   clock_t clock(void);

DESCRIPTION
   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
   The clock() function shall return the implementation's best approximation to the processor time
   used by the process since the beginning of an implementation-defined era related only to the
   process invocation.

RETURN VALUE
   To determine the time in seconds, the value returned by clock() should be divided by the value
   of the macro CLOCKS_PER_SEC. CLOCKS_PER_SEC is defined to be one million in <time.h>.
   If the processor time used is not available or its value cannot be represented, the function shall
   return the value (clock_t)-1.

ERRORS
   No errors are defined.

EXAMPLES
   None.

APPLICATION USAGE
   In order to measure the time spent in a program, clock() should be called at the start of the
   program and its return value subtracted from the value returned by subsequent calls. The value
   returned by clock() is defined for compatibility across systems that have clocks with different
   resolutions. The resolution on any particular system need not be to microsecond accuracy.
   The value returned by clock() may wrap around on some implementations. For example, on a
   machine with 32-bit values for clock_t, it wraps after 2 147 seconds or 36 minutes.

RATIONALE
   None.

FUTURE DIRECTIONS
   None.

SEE ALSO
   asctime(), ctime(), difftime(), gmtime(), localtime(), mktime(), strftime(), strptime(), time(), utime(),
   the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

CHANGE HISTORY
   First released in Issue 1. Derived from Issue 1 of the SVID.
NAME

clock_getcpuclockid — access a process CPU-time clock (ADVANCED REALTIME)

SYNOPSIS

CPT

```c
#include <time.h>

int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
```

DESCRIPTION

The `clock_getcpuclockid()` function shall return the clock ID of the CPU-time clock of the process specified by `pid`. If the process described by `pid` exists and the calling process has permission, the clock ID of this clock shall be returned in `clock_id`.

If `pid` is zero, the `clock_getcpuclockid()` function shall return the clock ID of the CPU-time clock of the process making the call, in `clock_id`.

The conditions under which one process has permission to obtain the CPU-time clock ID of other processes are implementation-defined.

RETURN VALUE

Upon successful completion, `clock_getcpuclockid()` shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The `clock_getcpuclockid()` function shall fail if:

- `[EPERM]` The requesting process does not have permission to access the CPU-time clock for the process.

The `clock_getcpuclockid()` function may fail if:

- `[ESRCH]` No process can be found corresponding to the process specified by `pid`.

EXAMPLES

None.

APPLICATION USAGE

The `clock_getcpuclockid()` function is part of the Process CPU-Time Clocks option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`clock_getres()`, `timer_create()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<time.h>`

CHANGE HISTORY


In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.
NAME

clock_getres, clock_gettime, clock_settime — clock and timer functions (REALTIME)

SYNOPSIS

#include <time.h>

int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);

DESCRIPTION

The clock_getres() function shall return the resolution of any clock. Clock resolutions are
implementation-defined and cannot be set by a process. If the argument res is not NULL, the
resolution of the specified clock shall be stored in the location pointed to by res. If res is NULL,
the clock resolution is not returned. If the time argument of clock_settime() is not a multiple of res,
then the value is truncated to a multiple of res.

The clock_gettime() function shall return the current value tp for the specified clock, clock_id.

The clock_settime() function shall set the specified clock, clock_id, to the value specified by tp.
Time values that are between two consecutive non-negative integer multiples of the resolution
of the specified clock shall be truncated down to the smaller multiple of the resolution.

A clock may be system-wide (that is, visible to all processes) or per-process (measuring time that
is meaningful only within a process). All implementations shall support a clock_id of
CLOCK_REALTIME as defined in <time.h>. This clock represents the realtime clock for the
system. For this clock, the values returned by clock_gettime() and specified by clock_settime()
represent the amount of time (in seconds and nanoseconds) since the Epoch. An implementation
may also support additional clocks. The interpretation of time values for these clocks is
unspecified.

If the value of the CLOCK_REALTIME clock is set via clock_settime(), the new value of the clock
shall be used to determine the time of expiration for absolute time services based upon the
CLOCK_REALTIME clock. This applies to the time at which armed absolute timers expire. If the
absolute time requested at the invocation of such a time service is before the new value of the
clock, the time service shall expire 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
threads that are blocked waiting for a relative time service based upon this clock, including the
nanosleep() function; nor on the expiration of relative timers based upon this clock.
Consequently, these time services shall expire when the requested relative interval elapses,
independently of the new or old value of the clock.

If the Monotonic Clock option is supported, all implementations shall support a clock_id of
CLOCK_MONOTONIC defined in <time.h>. This clock represents the monotonic clock for the
system. For this clock, the value returned by clock_gettime() represents the amount of time (in
seconds and nanoseconds) since an unspecified point in the past (for example, system start-up
time, or the Epoch). This point does not change after system start-up time. The value of the
CLOCK_MONOTONIC clock cannot be set via clock_settime(). This function shall fail if it is
invoked with a clock_id argument of CLOCK_MONOTONIC.

The effect of setting a clock via clock_settime() on armed per-process timers associated with a
clock other than CLOCK_REALTIME is implementation-defined.

If the value of the CLOCK_REALTIME clock is set via clock_settime(), the new value of the clock
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.

If _POSIX_CPUTIME is defined, implementations shall support clock ID values obtained by
invoking clock_getcpu_clockid(), 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)).

If _POSIX_THREAD_CPUTIME is defined, implementations shall support clock ID values
obtained by invoking pthread_getcpu_clockid(), 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)).

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.

ERRORS
The clock_getres(), clock_gettime(), and clock_settime() functions shall fail if:

[EINV AL] The clock_id argument does not specify a known clock.

The clock_settime() function shall fail if:

[EINV AL] The tp argument to clock_settime( ) is outside the range for the given clock ID.

[EINV AL] The tp argument specified a nanosecond value less than zero or greater than
or equal to 1 000 million.

[MONE] The value of the clock_id argument is CLOCK_MONOTONIC.

The clock_settime() function may fail if:

[EPERM] The requesting process does not have the appropriate privilege to set the
specified clock.
EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Timers option and need not be available on all implementations.

Note that the absolute value of the monotonic clock is meaningless (because its origin is
arbitrary), and thus there is no need to set it. Furthermore, realtime applications can rely on the
fact that the value of this clock is never set and, therefore, that time intervals measured with this
clock will not be affected by calls to clock_settime().

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getcpuclockid(), clock_nanosleep(), ctime(), mq_timedreceive(), mq_timedsend(), nanosleep(),
 pthread_mutex_timedlock(), sem_timedwait(), time(), timer_create(), timer_settime(), the Base
Definitions volume of IEEE Std 1003.1-2001, <time.h>

CHANGE HISTORY

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

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an
implementation does not support the Timers option.

The APPLICATION USAGE section is added.

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

• Clarification is added of the effect of resetting the clock resolution.

CPU-time clocks and the clock_getcpuclockid() function are added for alignment with

The following changes are added for alignment with IEEE Std 1003.1j-2000:

• The DESCRIPTION is updated as follows:
  — The value returned by clock_gettime() for CLOCK_MONOTONIC is specified.
  — The clock_settime() function failing for CLOCK_MONOTONIC is specified.
  — The effects of clock_settime() on the clock_nanosleep() function with respect to
    CLOCK_REALTIME is specified.

• An [EINVAL] error is added to the ERRORS section, indicating that clock_settime() fails for
  CLOCK_MONOTONIC.

• The APPLICATION USAGE section notes that the CLOCK_MONOTONIC clock need not
  and shall not be set by clock_settime() since the absolute value of the CLOCK_MONOTONIC
  clock is meaningless.

• The clock_nanosleep(), mq_timedreceive(), mq_timedsend(), pthread_mutex_timedlock(),
  sem_timedwait(), timer_create(), and timer_settime() functions are added to the SEE ALSO
  section.
NAME  
clock_nanosleep — high resolution sleep with specifiable clock (ADVANCED REALTIME)

SYNOPSIS

```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
const struct timespec *rqtp, struct timespec *rmtp);
```

DESCRIPTION

If the flag TIMER_ABSTIME is not set in the flags argument, the clock_nanosleep() function shall cause the current thread to be suspended from execution until either the time interval specified by the rqtp argument has elapsed, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. The clock used to measure the time shall be the clock specified by clock_id.

If the flag TIMER_ABSTIME is set in the flags argument, the clock_nanosleep() function shall cause the current thread to be suspended from execution until either the time value of the clock specified by clock_id reaches the absolute time specified by the rqtp argument, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. If, at the time of the call, the time value specified by rqtp is less than or equal to the time value of the specified clock, then clock_nanosleep() shall return immediately and the calling process shall not be suspended.

The suspension time caused by this function may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution, or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time for the relative clock_nanosleep() function (that is, with the TIMER_ABSTIME flag not set) shall not be less than the time interval specified by rqtp, as measured by the corresponding clock. The suspension for the absolute clock_nanosleep() function (that is, with the TIMER_ABSTIME flag set) shall be in effect at least until the value of the corresponding clock reaches the absolute time specified by rqtp, except for the case of being interrupted by a signal.

The use of the clock_nanosleep() function shall have no effect on the action or blockage of any signal.

The clock_nanosleep() function shall fail if the clock_id argument refers to the CPU-time clock of the calling thread. It is unspecified whether clock_id values of other CPU-time clocks are allowed.

RETURN VALUE

If the clock_nanosleep() function returns because the requested time has elapsed, its return value shall be zero.

If the clock_nanosleep() function returns because it has been interrupted by a signal, it shall return the corresponding error value. For the relative clock_nanosleep() function, if the rmtp argument is non-NULL, the timespec structure referenced by it shall be updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the rmtp argument is NULL, the remaining time is not returned. The absolute clock_nanosleep() function has no effect on the structure referenced by rmtp.

If clock_nanosleep() fails, it shall return the corresponding error value.
ERRORS

The `clock_nanosleep()` function shall fail if:

- `[EINTR]` The `clock_nanosleep()` function was interrupted by a signal.
- `[EINVAL]` The `rqtp` argument specified a nanosecond value less than zero or greater than or equal to 1,000 million; or the TIMER_ABSTIME flag was specified in flags and the `rqtp` argument is outside the range for the clock specified by `clock_id`; or the `clock_id` argument does not specify a known clock, or specifies the CPU-time clock of the calling thread.
- `[ENOTSUP]` The `clock_id` argument specifies a clock for which `clock_nanosleep()` is not supported, such as a CPU-time clock.

EXAMPLES

None.

APPLICATION USAGE

Calling `clock_nanosleep()` with the value TIMER_ABSTIME not set in the `flags` argument and with a `clock_id` of CLOCK_REALTIME is equivalent to calling `nanosleep()` with the same `rqtp` and `rmtp` arguments.

RATIONALE

The `nanosleep()` function specifies that the system-wide clock CLOCK_REALTIME is used to measure the elapsed time for this time service. However, with the introduction of the monotonic clock CLOCK_MONOTONIC a new relative sleep function is needed to allow an application to take advantage of the special characteristics of this clock.

There are many applications in which a process needs to be suspended and then activated multiple times in a periodic way; for example, to poll the status of a non-interrupting device or to refresh a display device. For these cases, it is known that precise periodic activation cannot be achieved with a relative `sleep()` or `nanosleep()` function call. Suppose, for example, a periodic process that is activated at time $T_0$, executes for a while, and then wants to suspend itself until time $T_0+T$, the period being $T$. If this process wants to use the `nanosleep()` function, it must first call `clock_gettime()` to get the current time, then calculate the difference between the current time and $T_0+T$ and, finally, call `nanosleep()` using the computed interval. However, the process could be preempted by a different process between the two function calls, and in this case the interval computed would be wrong; the process would wake up later than desired. This problem would not occur with the absolute `clock_nanosleep()` function, since only one function call would be necessary to suspend the process until the desired time. In other cases, however, a relative sleep is needed, and that is why both functionalities are required.

Although it is possible to implement periodic processes using the timers interface, this implementation would require the use of signals, and the reservation of some signal numbers. In this regard, the reasons for including an absolute version of the `clock_nanosleep()` function in IEEE Std 1003.1-2001 are the same as for the inclusion of the relative `nanosleep()`.

It is also possible to implement precise periodic processes using `pthread_cond_timedwait()`, in which an absolute timeout is specified that takes effect if the condition variable involved is never signaled. However, the use of this interface is unnatural, and involves performing other operations on mutexes and condition variables that imply an unnecessary overhead. Furthermore, `pthread_cond_timedwait()` is not available in implementations that do not support threads.

Although the interface of the relative and absolute versions of the new high resolution sleep service is the same `clock_nanosleep()` function, the `rmtp` argument is only used in the relative sleep. This argument is needed in the relative `clock_nanosleep()` function to reissue the function.
call if it is interrupted by a signal, but it is not needed in the absolute `clock_nanosleep()` function call; if the call is interrupted by a signal, the absolute `clock_nanosleep()` function can be invoked again with the same `rqtp` argument used in the interrupted call.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`clock_getres()`, `nanosleep()`, `pthread_cond_timedwait()`, `sleep()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<time.h>`

**CHANGE HISTORY**
NAME

clock_settime — clock and timer functions (REALTIME)

SYNOPSIS

#include <time.h>

int clock_settime(clockid_t clock_id, const struct timespec *tp);

DESCRIPTION

Refer to clock_getres().
NAME
clog, clogf, clogl — complex natural logarithm functions

SYNOPSIS
#include <complex.h>
double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);

DESCRIPTION
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

These functions shall compute the complex natural (base e) logarithm of z, with a branch cut
along the negative real axis.

RETURN VALUE
These functions shall return the complex natural logarithm value, in the range of a strip
mathematically unbounded along the real axis and in the interval [−iπ, +iπ] along the imaginary
axis.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
cexp(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY
NAME

close — close a file descriptor

SYNOPSIS

#include <unistd.h>

int close(int fildes);

DESCRIPTION

The close() function shall deallocate the file descriptor indicated by fildes. To deallocate means to make the file descriptor available for return by subsequent calls to open() or other functions that allocate file descriptors. All outstanding record locks owned by the process on the file associated with the file descriptor shall be removed (that is, unlocked).

If close() is interrupted by a signal that is to be caught, it shall return −1 with errno set to [EINTR] and the state of fildes is unspecified. If an I/O error occurred while reading from or writing to the file system during close(), it may return −1 with errno set to [EIO]; if this error is returned, the state of fildes is unspecified.

When all file descriptors associated with a pipe or FIFO special file are closed, any data remaining in the pipe or FIFO shall be discarded.

When all file descriptors associated with an open file description have been closed, the open file description shall be freed.

If the link count of the file is 0, when all file descriptors associated with the file are closed, the space occupied by the file shall be freed and the file shall no longer be accessible.

XSR If a STREAMS-based fildes is closed and the calling process was previously registered to receive a SIGPOLL signal for events associated with that STREAM, the calling process shall be unregistered for events associated with the STREAM. The last close() for a STREAM shall cause the STREAM associated with fildes to be dismantled. If O_NONBLOCK is not set and there have been no signals posted for the STREAM, and if there is data on the module's write queue, close() shall wait for an unspecified time (for each module and driver) for any output to drain before dismantling the STREAM. The time delay can be changed via an I_SETCLTIME ioctl() request. If the O_NONBLOCK flag is set, or if there are any pending signals, close() shall not wait for output to drain, and shall dismantle the STREAM immediately.

If the implementation supports STREAMS-based pipes, and fildes is associated with one end of a pipe, the last close() shall cause a hangup to occur on the other end of the pipe. In addition, if the other end of the pipe has been named by fattach(), then the last close() shall force the named end to be detached by fdetach(). If the named end has no open file descriptors associated with it and gets detached, the STREAM associated with that end shall also be dismantled.

XSI If fildes refers to the master side of a pseudo-terminal, and this is the last close, a SIGHUP signal shall be sent to the process group, if any, for which the slave side of the pseudo-terminal is the controlling terminal. It is unspecified whether closing the master side of the pseudo-terminal flushes all queued input and output.

XSR If fildes refers to the slave side of a STREAMS-based pseudo-terminal, a zero-length message may be sent to the master.

AIO When there is an outstanding cancelable asynchronous I/O operation against fildes when close() is called, that I/O operation may be canceled. An I/O operation that is not canceled completes as if the close() operation had not yet occurred. All operations that are not canceled shall complete as if the close() blocked until the operations completed. The close() operation itself need not block awaiting such I/O completion. Whether any I/O operation is canceled, and which I/O operation may be canceled upon close(), is implementation-defined.
System Interfaces

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.

If `fildes` 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.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned and `errno` set to indicate the error.

ERRORS

The `close()` function shall fail if:

- [EBADF] The `fildes` argument is not a valid file descriptor.
- [EINTR] The `close()` function was interrupted by a signal.
- The `close()` function may fail if:
  - [EIO] An I/O error occurred while reading from or writing to the file system.

EXAMPLES

Reassigning a File Descriptor

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.

```
#include <unistd.h>
...
int pfd;
...
close(1);
dup(pfd);
close(pfd);
...
```

Incidentally, this is exactly what could be achieved using:

```
dup2(pfd, 1);
close(pfd);
```

Closing a File Descriptor

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.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```
APPLICATION USAGE

An application that had used the stdio routine fopen() to open a file should use the corresponding fclose() routine rather than close(). Once a file is closed, the file descriptor no longer exists, since the integer corresponding to it no longer refers to a file.

RATIONALE

The use of interruptible device close routines should be discouraged to avoid problems with the implicit closes of file descriptors by exec and exit(). This volume of IEEE Std 1003.1-2001 only intends to permit such behavior by specifying the [EINTR] error condition.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.6 (on page 38), fattach(), fclose(), fdetach(), fopen(), ioctl(), open(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY

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

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Issue 6

The DESCRIPTION related to a STREAMS-based file or pseudo-terminal is marked as part of the XSI STREAMS Option Group.

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

- The [EIO] error condition is added as an optional error.
- The DESCRIPTION is updated to describe the state of the fildes file descriptor as unspecified if an I/O error occurs and an [EIO] error condition is returned.

Text referring to sockets is added to the DESCRIPTION.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that shared memory objects and memory mapped files (and not typed memory objects) are the types of memory objects to which the paragraph on last closes applies.
NAME
   closedir — close a directory stream

SYNOPSIS
   #include <dirent.h>
   int closedir(DIR *dirp);

DESCRIPTION
   The closedir() function shall close the directory stream referred to by the argument dirp. Upon
   return, the value of dirp may no longer point to an accessible object of the type DIR. If a file
   descriptor is used to implement type DIR, that file descriptor shall be closed.

RETURN VALUE
   Upon successful completion, closedir() shall return 0; otherwise, −1 shall be returned and errno
   set to indicate the error.

ERRORS
   The closedir() function may fail if:
   [EBADF] The dirp argument does not refer to an open directory stream.
   [EINVAL] The closedir() function was interrupted by a signal.

EXAMPLES
   Closing a Directory Stream
   The following program fragment demonstrates how the closedir() function is used.
   ...
   DIR *dir;
   struct dirent *dp;
   ...
   if ((dir = opendir (".")) == NULL) {
   ...
   } while ((dp = readdir (dir)) != NULL) {
   ...
   } while ((dp = readdir (dir)) != NULL) {
   ...
   closedir(dir);
   ...
   ...

APPLICATION USAGE
   None.

RATIONALE
   None.

FUTURE DIRECTIONS
   None.

SEE ALSO
   opendir(), the Base Definitions volume of IEEE Std 1003.1-2001, <dirent.h>
**CHANGE HISTORY**

First released in Issue 2.

**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.

- The [EINTR] error condition is added as an optional error condition.
NAME

closelog, openlog, setlogmask, syslog — control system log

SYNOPSIS

```c
#include <syslog.h>

void closelog(void);
void openlog(const char *ident, int logopt, int facility);
int setlogmask(int maskpri);
void syslog(int priority, const char *message, ... /* arguments */);
```

DESCRIPTION

The `syslog()` function shall send a message to an implementation-defined logging facility, which may log it in an implementation-defined system log, write it to the system console, forward it to a list of users, or forward it to the logging facility on another host over the network. The logged message shall include a message header and a message body. The message header contains at least a timestamp and a tag string.

The message body is generated from the `message` and following arguments in the same manner as if these were arguments to `printf()`, except that the additional conversion specification `%m` shall be recognized; it shall convert no arguments, shall cause the output of the error message string associated with the value of `errno` on entry to `syslog()`, and may be mixed with argument specifications of the "%n$s" form. If a complete conversion specification with the `m` conversion specifier character is not just `%m`, the behavior is undefined. A trailing `<newline>` may be added if needed.

Values of the `priority` argument are formed by OR’ing together a severity-level value and an optional facility value. If no facility value is specified, the current default facility value is used.

Possible values of severity level include:

- `LOG_EMERG` A panic condition.
- `LOG_ALERT` A condition that should be corrected immediately, such as a corrupted system database.
- `LOG_CRIT` Critical conditions, such as hard device errors.
- `LOG_ERR` Errors.
- `LOG_WARNING` Warning messages.
- `LOG_NOTICE` Conditions that are not error conditions, but that may require special handling.
- `LOG_INFO` Informational messages.
- `LOG_DEBUG` Messages that contain information normally of use only when debugging a program.

The facility indicates the application or system component generating the message. Possible facility values include:

- `LOG_USER` Messages generated by arbitrary processes. This is the default facility identifier if none is specified.
- `LOG_LOCAL0` Reserved for local use.
System Interfaces

closelog()

LOG_LOCAL1 Reserved for local use.
LOG_LOCAL2 Reserved for local use.
LOG_LOCAL3 Reserved for local use.
LOG_LOCAL4 Reserved for local use.
LOG_LOCAL5 Reserved for local use.
LOG_LOCAL6 Reserved for local use.
LOG_LOCAL7 Reserved for local use.

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:

- LOG_PID Log the process ID with each message. This is useful for identifying specific processes.
- LOG_CONS Write messages to the system console if they cannot be sent to the logging facility. The syslog() function ensures that the process does not acquire the console as a controlling terminal in the process of writing the message.
- LOG_NDELAY Open the connection to the logging facility immediately. Normally the open is delayed until the first message is logged. This is useful for programs that need to manage the order in which file descriptors are allocated.
- LOG_ODELAY Delay open until syslog() is called.
- LOG_NOWAIT Do not wait for child processes that may have been created during the course of logging the message. This option should be used by processes that enable notification of child termination using SIGCHLD, since syslog() may otherwise block waiting for a child whose exit status has already been collected.

The facility argument encodes a default facility to be assigned to all messages that do not have an explicit facility already encoded. The initial default facility is LOG_USER.

The openlog() and syslog() functions may allocate a file descriptor. It is not necessary to call openlog() prior to calling syslog().

The closelog() function shall close any open file descriptors allocated by previous calls to openlog() or syslog().

The setlogmask() function shall set the log priority mask for the current process to maskpri and return the previous mask. If the maskpri argument is 0, the current log mask is not modified. Calls by the current process to syslog() with a priority not set in maskpri shall be rejected. The default log mask allows all priorities to be logged. A call to openlog() is not required prior to calling setlogmask().

Symbolic constants for use as values of the logopt, facility, priority, and maskpri arguments are defined in the <syslog.h> header.

RETURN VALUE

The setlogmask() function shall return the previous log priority mask. The closelog(), openlog(), and syslog() functions shall not return a value.
closelog()

7104 ERRORS
7105 No errors are defined.

7106 EXAMPLES

7107 Using openlog()
7108 The following example causes subsequent calls to syslog() to log the process ID with each message, and to write messages to the system console if they cannot be sent to the logging facility.
7109
7110 #include <syslog.h>
7111 char *ident = "Process demo";
7112 int logopt = LOG_PID | LOG_CONS;
7113 int facility = LOG_USER;
7114 ...
7115 openlog(ident, logopt, facility);

7117 Using setlogmask()
7118 The following example causes subsequent calls to syslog() to accept error messages or messages generated by arbitrary processes, and to reject all other messages.
7119
7120 #include <syslog.h>
7121 int result;
7122 int mask = LOG_MASK (LOG_ERR | LOG_USER);
7123 ...
7124 result = setlogmask(mask);

7125 Using syslog
7126 The following example sends the message "This is a message" to the default logging facility, marking the message as an error message generated by random processes.
7127
7128 #include <syslog.h>
7129 char *message = "This is a message";
7130 int priority = LOG_ERR | LOG_USER;
7131 ...
7132 syslog(priority, message);

7133 APPLICATION USAGE
7134 None.

7135 RATIONALE
7136 None.

7137 FUTURE DIRECTIONS
7138 None.

7139 SEE ALSO
7140 printf(), the Base Definitions volume of IEEE Std 1003.1-2001, <syslog.h>
CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
NAME
confstr — get configurable variables

SYNOPSIS
#include <unistd.h>
size_t confstr(int name, char *buf, size_t len);

DESCRIPTION
The confstr() function shall return configuration-defined string values. Its use and purpose are
similar to sysconf(), but it is used where string values rather than numeric values are returned.

The name argument represents the system variable to be queried. The implementation shall
support the following name values, defined in <unistd.h>. It may support others:

_CS_PATH
_CS_POSIX_V6_ILP32_OFF32_CFLAGS
_CS_POSIX_V6_ILP32_OFF32_LDFLAGS
_CS_POSIX_V6_ILP32_OFF32_LIBS
_CS_POSIX_V6_ILP32_OFFBIG_CFLAGS
_CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS
_CS_POSIX_V6_ILP32_OFFBIG_LIBS
_CS_POSIX_V6_LP64_OFF64_CFLAGS
_CS_POSIX_V6_LP64_OFF64_LDFLAGS
_CS_POSIX_V6_LP64_OFF64_LIBS
_CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS
_CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS
_CS_POSIX_V6_LPBIG_OFFBIG_LIBS
_CS_POSIX_V6_WIDTH_RESTRICTED_ENVS

XSI
_CS_XBS5_ILP32_OFF32_CFLAGS (LEGACY)
_CS_XBS5_ILP32_OFF32_LDFLAGS (LEGACY)
_CS_XBS5_ILP32_OFF32_LIBS (LEGACY)
_CS_XBS5_ILP32_OFFBIG_CFLAGS (LEGACY)
_CS_XBS5_ILP32_OFFBIG_LDFLAGS (LEGACY)
_CS_XBS5_ILP32_OFFBIG_LIBS (LEGACY)
_CS_XBS5_LP64_OFF64_CFLAGS (LEGACY)
_CS_XBS5_LP64_OFF64_LDFLAGS (LEGACY)
_CS_XBS5_LP64_OFF64_LIBS (LEGACY)
_CS_XBS5_LPBIG_OFFBIG_CFLAGS (LEGACY)
_CS_XBS5_LPBIG_OFFBIG_LDFLAGS (LEGACY)
_CS_XBS5_LPBIG_OFFBIG_LIBS (LEGACY)
_CS_XBS5_LPBIG_OFFBIG_LINTFLAGS (LEGACY)

If len is not 0, and if name has a configuration-defined value, confstr() shall copy that value into
the len-byte buffer pointed to by buf. If the string to be returned is longer than len bytes,
including the terminating null, then confstr() shall truncate the string to len–1 bytes and null-
terminate the result. The application can detect that the string was truncated by comparing the
value returned by confstr() with len.

If len is 0 and buf is a null pointer, then confstr() shall still return the integer value as defined
below, but shall not return a string. If len is 0 but buf is not a null pointer, the result is
unspecified.
If the implementation supports the POSIX shell option, the string stored in `buf` after a call to:

```c
confstr(_CS_PATH, buf, sizeof(buf))
```

can be used as a value of the `PATH` environment variable that accesses all of the standard utilities of IEEE Std 1003.1-2001, if the return value is less than or equal to `sizeof(buf)`.

**RETURN VALUE**

If `name` has a configuration-defined value, `confstr()` shall return the size of buffer that would be needed to hold the entire configuration-defined value including the terminating null. If this return value is greater than `len`, the string returned in `buf` is truncated.

If `name` is invalid, `confstr()` shall return 0 and set `errno` to indicate the error.

If `name` does not have a configuration-defined value, `confstr()` shall return 0 and leave `errno` unchanged.

**ERRORS**

The `confstr()` function shall fail if:

- `[EINVAL]` The value of the `name` argument is invalid.

**EXAMPLES**

None.

**APPLICATION USAGE**

An application can distinguish between an invalid `name` parameter value and one that corresponds to a configurable variable that has no configuration-defined value by checking if `errno` is modified. This mirrors the behavior of `sysconf()`.

The original need for this function was to provide a way of finding the configuration-defined default value for the environment variable `PATH`. Since `PATH` can be modified by the user to include directories that could contain utilities replacing the standard utilities in the Shell and Utilities volume of IEEE Std 1003.1-2001, applications need a way to determine the system-supplied `PATH` environment variable value that contains the correct search path for the standard utilities.

An application could use:

```c
confstr(name, (char *)NULL, (size_t)0)
```

to find out how big a buffer is needed for the string value; use `malloc()` to allocate a buffer to hold the string; and call `confstr()` again to get the string. Alternately, it could allocate a fixed, static buffer that is big enough to hold most answers (perhaps 512 or 1024 bytes), but then use `malloc()` to allocate a larger buffer if it finds that this is too small.

**RATIONALE**

Application developers can normally determine any configuration variable by means of reading from the stream opened by a call to:

```c
popen("command -p getconf variable", "r");
```

The `confstr()` function with a `name` argument of `_CS_PATH` returns a string that can be used as a `PATH` environment variable setting that will reference the standard shell and utilities as described in the Shell and Utilities volume of IEEE Std 1003.1-2001.

The `confstr()` function copies the returned string into a buffer supplied by the application instead of returning a pointer to a string. This allows a cleaner function in some implementations (such as those with lightweight threads) and resolves questions about when the application must copy the string returned.
FUTURE DIRECTIONS

None.

SEE ALSO

CHANGE HISTORY

Issue 5
A table indicating the permissible values of name is added to the DESCRIPTION. All those marked EX are new in this issue.

Issue 6
The Open Group Corrigendum U033/7 is applied. The return value for the case returning the size of the buffer now explicitly states that this includes the terminating null.

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

• The DESCRIPTION is updated with new arguments which can be used to determine configuration strings for C compiler flags, linker/loader flags, and libraries for each different supported programming environment. This is a change to support data size neutrality.

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

• The DESCRIPTION is updated to include text describing how _CS_PATH can be used to obtain a PATH to access the standard utilities.

The macros associated with the c89 programming models are marked LEGACY and new equivalent macros associated with c99 are introduced.
NAME
conj, conjf, conjl — complex conjugate functions

SYNOPSIS
#include <complex.h>

double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);

DESCRIPTION
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.

These functions shall compute the complex conjugate of z, by reversing the sign of its imaginary part.

RETURN VALUE
These functions return the complex conjugate value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
carg(), cimag(), cproj(), creal(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY
connect()

NAME
connect — connect a socket

SYNOPSIS
#include <sys/socket.h>

int connect(int socket, const struct sockaddr *address,
            socklen_t address_len);

DESCRIPTION
The connect() function shall attempt to make a connection on a socket. The function takes the
following arguments:

socket Specifies the file descriptor associated with the socket.
address Points to a sockaddr structure containing the peer address. The length and
format of the address depend on the address family of the socket.
address_len Specifies the length of the sockaddr structure pointed to by the address
argument.

If the socket has not already been bound to a local address, connect() shall bind it to an address
which, unless the socket’s address family is AF_UNIX, is an unused local address.

If the initiating socket is not connection-mode, then connect() shall set the socket’s peer address,
and no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all
datagrams are sent on subsequent send() functions, and limits the remote sender for subsequent
recv() functions. If address is a null address for the protocol, the socket’s peer address shall be
reset.

If the initiating socket is connection-mode, then connect() shall attempt to establish a connection
to the address specified by the address argument. If the connection cannot be established
immediately and O_NONBLOCK is not set for the file descriptor for the socket, connect() shall
block for up to an unspecified timeout interval until the connection is established. If the timeout
interval expires before the connection is established, connect() shall fail and the connection
attempt shall be aborted. If connect() is interrupted by a signal that is caught while blocked
waiting to establish a connection, connect() shall fail and set errno to [EINTR], but the connection
request shall not be aborted, and the connection shall be established asynchronously.

If the connection cannot be established immediately and O_NONBLOCK is set for the file
descriptor for the socket, connect() shall fail and set errno to [EINPROGRESS], but the connection
request shall not be aborted, and the connection shall be established asynchronously.

Subsequent calls to connect() for the same socket, before the connection is established, shall fail
and set errno to [EALREADY].

When the connection has been established asynchronously, select() and poll() shall indicate that
the file descriptor for the socket is ready for writing.

The socket in use may require the process to have appropriate privileges to use the connect() function.

RETURN VALUE
Upon successful completion, connect() shall return 0; otherwise, −1 shall be returned and errno
set to indicate the error.

ERRORS
The connect() function shall fail if:

[EADDRNOTAVAIL]
The specified address is not available from the local machine.
connect() function may fail if:

- [EACCES] Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
- [EADDRINUSE] Attempt to establish a connection that uses addresses that are already in use.
- [ECONNRESET] Remote host reset the connection request.
- [EHOSTUNREACH] The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
- [EINVAL] The address family in the sockaddr structure.
- [EINVAL] The address_len argument is not a valid length for the address family; or
- [ENOENT] A component of the path name does not name an existing file or the path name is an empty string.
- [ENOTDIR] A component of the path prefix of the path name in address is not a directory.

If the address family of the socket is AF_UNIX, then connect() shall fail if:

- [EIO] An I/O error occurred while reading from or writing to the file system.
- [ELOOP] A loop exists in symbolic links encountered during resolution of the pathname in address.
- [EADDRINUSE] Attempt to establish a connection that uses addresses that are already in use.
connect()  

[ELOOP] More than [SYMLOOP_MAX] symbolic links were encountered during resolution of the pathname in address.

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds [PATH_MAX].

[ENETDOWN] The local network interface used to reach the destination is down.

[ENOBUFS] No buffer space is available.

[EOPNOTSUPP] The socket is listening and cannot be connected.

EXAMPLES

None.

APPLICATION USAGE

If connect() fails, the state of the socket is unspecified. Conforming applications should close the file descriptor and create a new socket before attempting to reconnect.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

accept(), bind(), close(), getsockname(), poll(), select(), send(), shutdown(), socket(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

CHANGE HISTORY

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

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.
NAME
copysign, copysignf, copysignl — number manipulation function

SYNOPSIS
#include <math.h>
double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);

DESCRIPTION
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

These functions shall produce a value with the magnitude of \(x\) and the sign of \(y\). On
implementations that represent a signed zero but do not treat negative zero consistently in
arithmetic operations, these functions regard the sign of zero as positive.

RETURN VALUE
Upon successful completion, these functions shall return a value with the magnitude of \(x\) and
the sign of \(y\).

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
\texttt{signbit()}, the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<math.h>}

CHANGE HISTORY
NAME

cos, cosf, cosl — cosine function

SYNOPSIS

#include <math.h>

double cos(double x);

float cosf(float x);

long double cosl(long double x);

DESCRIPTION

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.

These functions shall compute the cosine of their argument \( x \), measured in radians.

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

RETURN VALUE

Upon successful completion, these functions shall return the cosine of \( x \).

If \( x \) is NaN, a NaN shall be returned.

If \( x \) is \( \pm 0 \), the value 1.0 shall be returned.

If \( x \) is \( \pm \infty \), a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

- If the integer expression (\( \text{math_errhandling} \) & MATH_ERRNO) is non-zero, then \( \text{errno} \) shall be set to \[ \text{EDOM} \]. If the integer expression (\( \text{math_errhandling} \) & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

- If the integer expression (\( \text{math_errhandling} \) & MATH_ERRNO) is non-zero, the \( \text{domain error} \) shall occur.

EXAMPLES

Taking the Cosine of a 45-Degree Angle

```c
#include <math.h>
...
double radians = 45 * M_PI / 180;
double result;
...
result = cos(radians);
```

APPLICATION USAGE

These functions may lose accuracy when their argument is near an odd multiple of \( \pi/2 \) or is far from 0.

On error, the expressions (\( \text{math_errhandling} \) & MATH_ERRNO) and (\( \text{math_errhandling} \) & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.
RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
acos(), feclearexcept(), fetestexcept(), isnan(), sin(), tan(), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The cosf() and cosl() functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.
cosh()

NAME

cosh, coshf, coshl — hyperbolic cosine functions

SYNOPSIS

#include <math.h>

double cosh(double x);
float coshf(float x);
long double coshl(long double x);

DESCRIPTION

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

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

An application wishing to check for error situations should set errno to zero and call
fearccept(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.

RETURN VALUE

Upon successful completion, these functions shall return the hyperbolic cosine of x.

If the correct value would cause overflow, a range error shall occur and cosh(), coshf(), and
coshl() shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL,
respectively.

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

If x is ±0, the value 1.0 shall be returned.

If x is ±Inf, +Inf shall be returned.

ERRORS

These functions shall fail if:

Range Error
The result would cause an overflow.

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.

EXAMPLES

None.

APPLICATION USAGE

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.

For IEEE Std 754-1985 double, 710.5 < |x| implies that cosh(x) has overflowed.

RATIONALE

None.

FUTURE DIRECTIONS

None.
SEE ALSO

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

CHANGE HISTORY

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

Issue 5

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

Issue 6

The coshf() and coshl() functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
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.
NAME
    cosl — cosine function

SYNOPSIS
    #include <math.h>
    long double cosl(long double x);

DESCRIPTION
    Refer to cos().

NAME

cpow, cpowf, cpowl — complex power functions

SYNOPSIS

#include <complex.h>

double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x,
long double complex y);

DESCRIPTION

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.

These functions shall compute the complex power function \( x^y \), with a branch cut for the first parameter along the negative real axis.

RETURN VALUE

These functions shall return the complex power function value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cabs(), csqrt(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY

NAME
cproj, cprojf, cprojl — complex projection functions

SYNOPSIS
#include <complex.h>
double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);

DESCRIPTION
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.

These functions shall compute a projection of z onto the Riemann sphere: z projects to z, except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If z has an infinite part, then cproj(z) shall be equivalent to:

INFINITY + I * copysign(0.0, cimag(z))

RETURN VALUE
These functions shall return the value of the projection onto the Riemann sphere.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
Two topologies are commonly used in complex mathematics: the complex plane with its continuum of infinities, and the Riemann sphere with its single infinity. The complex plane is better suited for transcendental functions, the Riemann sphere for algebraic functions. The complex types with their multiplicity of infinities provide a useful (though imperfect) model for the complex plane. The cproj() function helps model the Riemann sphere by mapping all infinities to one, and should be used just before any operation, especially comparisons, that might give spurious results for any of the other infinities. Note that a complex value with one infinite part and one NaN part is regarded as an infinity, not a NaN, because if one part is infinite, the complex value is infinite independent of the value of the other part. For the same reason, cabs() returns an infinity if its argument has an infinite part and a NaN part.

FUTURE DIRECTIONS
None.

SEE ALSO
carg(), cimag(), conj(), creal(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY
NAME
creal, crealf, creall — complex real functions

SYNOPSIS
#include <complex.h>

double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);

DESCRIPTION
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.

These functions shall compute the real part of z.

RETURN VALUE
These functions shall return the real part value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
For a variable z of type complex:

z == creal(z) + cimag(z)*I

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
carg(), cimag(), conj(), cproj(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY
NAME
creat — create a new file or rewrite an existing one

SYNOPSIS
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *path, mode_t mode);

DESCRIPTION
The function call:
creat(path, mode)
shall be equivalent to:
open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)

RETURN VALUE
Refer to open().

ERRORS
Refer to open().

EXAMPLES
Creating a File
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.

#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = creat(filename, mode);

APPLICATION USAGE
None.

RATIONALE
The creat() function is redundant. Its services are also provided by the open() function. It has
been included primarily for historical purposes since many existing applications depend on it. It
is best considered a part of the C binding rather than a function that should be provided in other
languages.

FUTURE DIRECTIONS
None.

SEE ALSO
open(), the Base Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>, <sys/stat.h>,
<sys/types.h>
CHANGE HISTORY

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

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.
NAME
crypt — string encoding function (CRYPT)

SYNOPSIS
XSI
#include <unistd.h>

crypt ( const char *key, const char *salt );

DESCRIPTION
The crypt() function is a string encoding function. The algorithm is implementation-defined.
The key argument points to a string to be encoded. The salt argument is a string chosen from the set:

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789./
The first two characters of this string may be used to perturb the encoding algorithm.
The return value of crypt() points to static data that is overwritten by each call.
The crypt() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE
Upon successful completion, 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.

ERRORS
The crypt() function shall fail if:

[ENOSYS] The functionality is not supported on this implementation.

EXAMPLES
Encoding Passwords
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.
The putpwent() function, used in the following example, is not part of IEEE Std 1003.1-2001.

#include <unistd.h>
#include <pwd.h>
#include <string.h>
#include <stdio.h>
...
int valid_change;
int pfd; /* Integer for file descriptor returned by open(). */
FILE *fpfd; /* File pointer for use in putpwent(). */
struct passwd *p;
char user[100];
char oldpasswd[100];
char newpasswd[100];
char savepasswd[100];
...
valid_change = 0;
while ((p = getpwent()) != NULL) {
    /* Change entry if found. */
    if (strcmp(p->pw_name, user) == 0) {
        if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
            strcpy(savepasswd, crypt(newpasswd, user));
            p->pw_passwd = savepasswd;
            valid_change = 1;
        }
        else {
            fprintf(stderr, "Old password is not valid\n");
        }
    }
    /* Put passwd entry into ptmp. */
    putpwent(p, fpfd);
}

APPLICATION USAGE
The values returned by this function need not be portable among XSI-conformant systems.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
encrypt(), setkey(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.
NAME

csin, csinf, csinl — complex sine functions

SYNOPSIS

#include <complex.h>

double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);

DESCRIPTION

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.

These functions shall compute the complex sine of z.

RETURN VALUE

These functions shall return the complex sine value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

casin(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY

NAME
    csinh, csinhf, csinhl — complex hyperbolic sine functions

SYNOPSIS
    #include <complex.h>
    double complex csinh(double complex z);
    float complex csinhf(float complex z);
    long double complex csinhl(long double complex z);

DESCRIPTION
    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.
    These functions shall compute the complex hyperbolic sine of \( z \).

RETURN VALUE
    These functions shall return the complex hyperbolic sine value.

ERRORS
    No errors are defined.

EXAMPLES
    None.

APPLICATION USAGE
    None.

RATIONALE
    None.

FUTURE DIRECTIONS
    None.

SEE ALSO
    casinh(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY
NAME
    csinl — complex sine functions

SYNOPSIS
    #include <complex.h>
    long double complex csinl(long double complex z);

DESCRIPTION
    Refer to csin().
NAME

csqrt, csqrtf, csqrtl — complex square root functions

SYNOPSIS

#include <complex.h>

double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);

DESCRIPTION

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.

These functions shall compute the complex square root of z, with a branch cut along the negative real axis.

RETURN VALUE

These functions shall return the complex square root value, in the range of the right half-plane (including the imaginary axis).

ERRORS

No errors are defined.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cabs(), cpow(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY

ctan() System Interfaces

NAME
ctan, ctanf, ctanl — complex tangent functions

SYNOPSIS
#include <complex.h>

double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);

DESCRIPTION
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.

These functions shall compute the complex tangent of z.

RETURN VALUE
These functions shall return the complex tangent value.

ERRORS
No errors are defined.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
catan(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY
NAME
canh, ctanhf, ctanhl — complex hyperbolic tangent functions

SYNOPSIS
#include <complex.h>

double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);

DESCRIPTION
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

These functions shall compute the complex hyperbolic tangent of z.

RETURN VALUE
These functions shall return the complex hyperbolic tangent value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
ctanh(), the Base Definitions volume of IEEE Std 1003.1-2001, <complex.h>

CHANGE HISTORY
NAME
ctanl — complex tangent functions

SYNOPSIS
#include <complex.h>

long double complex ctanl(long double complex z);

DESCRIPTION
Refer to ctan().
NAME
ctermid — generate a pathname for the controlling terminal

SYNOPSIS
#include <stdio.h>
char *ctermid(char *s);

DESCRIPTION
The ctermid() function shall generate a string that, when used as a pathname, refers to the current controlling terminal for the current process. If ctermid() returns a pathname, access to the file is not guaranteed.

If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS functions, it shall ensure that the ctermid() function is called with a non-NULL parameter.

RETURN VALUE
If s is a null pointer, the string shall be generated in an area that may be static (and therefore may be overwritten by each call), the address of which shall be returned. Otherwise, s is assumed to point to a character array of at least L_ctermid bytes; the string is placed in this array and the value of s shall be returned. The symbolic constant L_ctermid is defined in <stdio.h>, and shall have a value greater than 0.

The ctermid() function shall return an empty string if the pathname that would refer to the controlling terminal cannot be determined, or if the function is unsuccessful.

ERRORS
No errors are defined.

EXAMPLES
Determining the Controlling Terminal for the Current Process
The following example returns a pointer to a string that identifies the controlling terminal for the current process. The pathname for the terminal is stored in the array pointed to by the ptr argument, which has a size of L_ctermid bytes, as indicated by the term argument.

#include <stdio.h>
...
char term[L_ctermid];
char *ptr;
ptr = ctermid(term);

APPLICATION USAGE
The difference between ctermid() and ttyname() is that ttyname() must be handed a file descriptor and return a path of the terminal associated with that file descriptor, while ctermid() returns a string (such as "/dev/tty") that refers to the current controlling terminal if used as a pathname.

RATIONALE
L_ctermid must be defined appropriately for a given implementation and must be greater than zero so that array declarations using it are accepted by the compiler. The value includes the terminating null byte.

Conforming applications that use threads cannot call ctermid() with NULL as the parameter if either _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS is defined. If s is not NULL, the ctermid() function generates a string that, when used as a pathname, refers to the
current controlling terminal for the current process. If s is NULL, the return value of ctermid() is undefined.

There is no additional burden on the programmer—changing to use a hypothetical thread-safe version of ctermid() along with allocating a buffer is more of a burden than merely allocating a buffer. Application code should not assume that the returned string is short, as some implementations have more than two pathname components before reaching a logical device name.

FUTURE DIRECTIONS

None.

SEE ALSO
ttyname(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY

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

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
ctime, ctime_r — convert a time value to a date and time string

SYNOPSIS
#include <time.h>

char *ctime(const time_t *clock);
TSF char *ctime_r(const time_t *clock, char *buf);

DESCRIPTION
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.

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:

asctime(localtime(clock))

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.

The ctime() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

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.

Unlike ctime(), the thread-safe version ctime_r() is not required to set tzname.

RETURN VALUE
The ctime() function shall return the pointer returned by asctime() with that broken-down time as an argument.
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.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
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.

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.

RATIONALE
None.
#### FUTURE DIRECTIONS

None.

#### SEE ALSO

`asctime()`, `clock()`, `difftime()`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `strptime()`, `time()`, `utime()`,

the Base Definitions volume of IEEE Std 1003.1-2001, `<time.h>`

#### CHANGE HISTORY

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

**Issue 5**

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

The `ctime_r()` function is included for alignment with the POSIX Threads Extension.

A note indicating that the `ctime()` function need not be reentrant is added to the DESCRIPTION.

**Issue 6**

Extensions beyond the ISO C standard are marked.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.
NAME

daylight — daylight savings time flag

SYNOPSIS

```c
#include <time.h>

extern int daylight;
```

DESCRIPTION

Refer to `tzset()`.
NAME
dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, dbm_nextkey,
dbm_open, dbm_store — database functions

SYNOPSIS
#include <ndbm.h>

int dbm_clearerr(DBM *db);
void dbm_close(DBM *db);
int dbm_delete(DBM *db, datum key);
int dbm_error(DBM *db);
datum dbm_fetch(DBM *db, datum key);
datum dbm_firstkey(DBM *db);
datum dbm_nextkey(DBM *db);
DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);
int dbm_store(DBM *db, datum key, datum content, int store_mode);

DESCRIPTION
These functions create, access, and modify a database.

A datum consists of at least two members, dptr and dsize. The dptr member points to an object
that is dsize bytes in length. Arbitrary binary data, as well as character strings, may be stored in
the object pointed to by dptr.

The database is stored in two files. One file is a directory containing a bitmap of keys and has
.dir as its suffix. The second file contains all data and has.pag as its suffix.

The dbm_open() function shall open a database. The file argument to the function is the
pathname of the database. The function opens two files named file.dir and file.pag. The
open_flags argument has the same meaning as the flags argument of open() except that a database
opened for write-only access opens the files for read and write access and the behavior of the
O_APPEND flag is unspecified. The file_mode argument has the same meaning as the third
argument of open().

The dbm_close() function shall close a database. The application shall ensure that argument db is
a pointer to a dbm structure that has been returned from a call to dbm_open().

These database functions shall support an internal block size large enough to support
key/content pairs of at least 1 023 bytes.

The dbm_fetch() function shall read a record from a database. The argument db is a pointer to a
database structure that has been returned from a call to dbm_open(). The argument key is a
datum that has been initialized by the application to the value of the key that matches the key of
the record the program is fetching.

The dbm_store() function shall write a record to a database. The argument db is a pointer to a
database structure that has been returned from a call to dbm_open(). The argument key is a
datum that has been initialized by the application to the value of the key that identifies (for
subsequent reading, writing, or deleting) the record the application is writing. The argument
content is a datum that has been initialized by the application to the value of the record the
program is writing. The argument store_mode controls whether dbm_store() replaces any pre-
existing record that has the same key that is specified by the key argument. The application shall
set store_mode to either DBM_INSERT or DBM_REPLACE. If the database contains a record that
matches the key argument and store_mode is DBM_REPLACE, the existing record shall be
replaced with the new record. If the database contains a record that matches the key argument
and store_mode is DBM_INSERT, the existing record shall be left unchanged and the new record
ignored. If the database does not contain a record that matches the `key` argument and `store_mode` is either DBM_INSERT or DBM_REPLACE, the new record shall be inserted in the database.

If the sum of a key/content pair exceeds the internal block size, the result is unspecified. Moreover, the application shall ensure that all key/content pairs that hash together fit on a single block. The `dbm_store()` function shall return an error in the event that a disk block fills with inseparable data.

The `dbm_delete()` function shall delete a record and its `key` from the database. The argument `db` is a pointer to a database structure that has been returned from a call to `dbm_open()`. The argument `key` is a `datum` that has been initialized by the application to the value of the key that identifies the record the program is deleting.

The `dbm_firstkey()` function shall return the first key in the database. The argument `db` is a pointer to a database structure that has been returned from a call to `dbm_open()`.

The `dbm_nextkey()` function shall return the next key in the database. The argument `db` is a pointer to a database structure that has been returned from a call to `dbm_open()`. The application shall ensure that the `dbm_firstkey()` function is called before calling `dbm_nextkey()`. Subsequent calls to `dbm_nextkey()` return the next key until all of the keys in the database have been returned.

The `dbm_error()` function shall return the error condition of the database. The argument `db` is a pointer to a database structure that has been returned from a call to `dbm_open()`.

The `dbm_clearerr()` function shall clear the error condition of the database. The argument `db` is a pointer to a database structure that has been returned from a call to `dbm_open()`.

The `dptr` pointers returned by these functions may point into static storage that may be changed by subsequent calls.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

**RETURN VALUE**

The `dbm_store()` and `dbm_delete()` functions shall return 0 when they succeed and a negative value when they fail.

The `dbm_store()` function shall return 1 if it is called with a `flags` value of DBM_INSERT and the function finds an existing record with the same key.

The `dbm_error()` function shall return 0 if the error condition is not set and return a non-zero value if the error condition is set.

The return value of `dbm_clearerr()` is unspecified.

The `dbm_firstkey()` and `dbm_nextkey()` functions shall return a key `datum`. When the end of the database is reached, the `dptr` member of the key is a null pointer. If an error is detected, the `dptr` member of the key shall be a null pointer and the error condition of the database shall be set.

The `dbm_fetch()` function shall return a content `datum`. If no record in the database matches the key or if an error condition has been detected in the database, the `dptr` member of the content shall be a null pointer.

The `dbm_open()` function shall return a pointer to a database structure. If an error is detected during the operation, `dbm_open()` shall return a `(DBM *)0`.

---

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
**dbm_clearerr()**

**ERRORS**
No errors are defined.

**EXAMPLES**
None.

**APPLICATION USAGE**

The following code can be used to traverse the database:

```c
for(key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

The `dbm_*` functions provided in this library should not be confused in any way with those of a general-purpose database management system. These functions do not provide for multiple search keys per entry, they do not protect against multi-user access (in other words they do not lock records or files), and they do not provide the many other useful database functions that are found in more robust database management systems. Creating and updating databases by use of these functions is relatively slow because of data copies that occur upon hash collisions. These functions are useful for applications requiring fast lookup of relatively static information that is to be indexed by a single key.

Note that a strictly conforming application is extremely limited by these functions: since there is no way to determine that the keys in use do not all hash to the same value (although that would be rare), a strictly conforming application cannot be guaranteed that it can store more than one block’s worth of data in the database. As long as a key collision does not occur, additional data may be stored, but because there is no way to determine whether an error is due to a key collision or some other error condition (`dbm_error()` being effectively a Boolean), once an error is detected, the application is effectively limited to guessing what the error might be if it wishes to continue using these functions.

The `dbm_delete()` function need not physically reclaim file space, although it does make it available for reuse by the database.

After calling `dbm_store()` or `dbm_delete()` during a pass through the keys by `dbm_firstkey()` and `dbm_nextkey()`, the application should reset the database by calling `dbm_firstkey()` before again calling `dbm_nextkey()`. The contents of these files are unspecified and may not be portable.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`open()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<ndbm.h>`

**CHANGE HISTORY**
First released in Issue 4, Version 2.

**Issue 5**
Moved from X/OPEN UNIX extension to BASE.

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

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
difftime — compute the difference between two calendar time values

SYNOPSIS
#include <time.h>
double difftime(time_t time1, time_t time0);

DESCRIPTION
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 difftime() function shall compute the difference between two calendar times (as returned by time()): time1−time0.

RETURN VALUE
The difftime() function shall return the difference expressed in seconds as a type double.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
asctime(), clock(), ctime(), gmtime(), localtime(), mktime(), strftime(), strptime(), time(), utime(), the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

CHANGE HISTORY
First released in Issue 4. Derived from the ISO C standard.
dirname — report the parent directory name of a file pathname

The dirname() function shall take a pointer to a character string that contains a pathname, and return a pointer to a string that is a pathname of the parent directory of that file. Trailing ‘/’ characters in the path are not counted as part of the path.

If path does not contain a ‘/’, then dirname() shall return a pointer to the string "..". If path is a null pointer or points to an empty string, dirname() shall return a pointer to the string ".".

The dirname() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The dirname() function shall return a pointer to a string that is the parent directory of path. If path is a null pointer or points to an empty string, a pointer to a string "." is returned.

The dirname() function may modify the string pointed to by path, and may return a pointer to static storage that may then be overwritten by subsequent calls to dirname().

No errors are defined.

The following code fragment reads a pathname, changes the current working directory to the parent directory, and opens the file.

```c
char path[PATH_MAX], *pathcopy;
int fd;
fgets(path, PATH_MAX, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

In the following table, the input string is the value pointed to by path, and the output string is the return value of the dirname() function.

<table>
<thead>
<tr>
<th>Input String</th>
<th>Output String</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;/usr/lib&quot;</td>
<td>&quot;/usr&quot;</td>
</tr>
<tr>
<td>&quot;/usr/&quot;</td>
<td>&quot;/&quot;</td>
</tr>
<tr>
<td>&quot;usr&quot;</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td>&quot;/&quot;</td>
<td>&quot;/&quot;</td>
</tr>
<tr>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td>&quot;..&quot;</td>
<td>&quot;.&quot;</td>
</tr>
</tbody>
</table>
Changing the Current Directory to the Parent Directory

The following program fragment reads a pathname, changes the current working directory to the parent directory, and opens the file.

```c
#include <unistd.h>
#include <limits.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <libgen.h>
...
char path[PATH_MAX], *pathcopy;
int fd;
...
fgets(path, PATH_MAX, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

**APPLICATION USAGE**

The `dirname()` and `basename()` functions together yield a complete pathname. The expression `dirname(path)` obtains the pathname of the directory where `basename(path)` is found.

Since the meaning of the leading "/" is implementation-defined, `dirname("//foo")` may return either "/" or "/" (but nothing else).

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`basename()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<libgen.h>`

**CHANGE HISTORY**

First released in Issue 4, Version 2.

**Issue 5**

Moved from X/OPEN UNIX extension to BASE.

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

A note indicating that this function need not be reentrant is added to the DESCRIPTION.
NAME

div — compute the quotient and remainder of an integer division

SYNOPSIS

#include <stdlib.h>

div_t div(int numer, int denom);

DESCRIPTION

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 div() function shall compute the quotient and remainder of the division of the numerator numer by the denominator denom. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, quot*denom+rem shall equal numer.

RETURN VALUE

The div() function shall return a structure of type div_t, comprising both the quotient and the remainder. The structure includes the following members, in any order:

int quot; /* quotient */
int rem;    /* remainder */

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ldiv(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.
NAME
dlclose — close a dlopen() object

SYNOPSIS

```c
#include <dlfcn.h>

int dlclose(void *handle);
```

DESCRIPTION

The `dlclose()` function shall inform the system that the object referenced by a `handle` returned from a previous `dlopen()` invocation is no longer needed by the application.

The use of `dlclose()` reflects a statement of intent on the part of the process, but does not create any requirement upon the implementation, such as removal of the code or symbols referenced by `handle`. Once an object has been closed using `dlclose()` an application should assume that its symbols are no longer available to `dlsym()`. All objects loaded automatically as a result of invoking `dlopen()` on the referenced object shall also be closed if this is the last reference to it.

Although a `dlclose()` operation is not required to remove structures from an address space, neither is an implementation prohibited from doing so. The only restriction on such a removal is that no object shall be removed to which references have been relocated, until or unless all such references are removed. For instance, an object that had been loaded with a `dlopen()` operation specifying the RTLD_GLOBAL flag might provide a target for dynamic relocations performed in the processing of other objects—in such environments, an application may assume that no relocation, once made, shall be undone or remade unless the object requiring the relocation has itself been removed.

RETURN VALUE

If the referenced object was successfully closed, `dlclose()` shall return 0. If the object could not be closed, or if `handle` does not refer to an open object, `dlclose()` shall return a non-zero value. More detailed diagnostic information shall be available through `dlerror()`.

ERRORS

No errors are defined.

EXAMPLES

The following example illustrates use of `dlopen()` and `dlclose()`:

```c
... /* Open a dynamic library and then close it ... */
#include <dlfcn.h>
void *mylib;
int eret;
mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
... eret = dlclose(mylib);
...```

APPLICATION USAGE

A conforming application should employ a `handle` returned from a `dlopen()` invocation only within a given scope bracketed by the `dlopen()` and `dlclose()` operations. Implementations are free to use reference counting or other techniques such that multiple calls to `dlopen()` referencing the same object may return the same object for `handle`. Implementations are also free to reuse a `handle`. For these reasons, the value of a `handle` must be treated as an opaque object by the application, used only in calls to `dlsym()` and `dlclose()`.
**Rationale**

None.

**Future Directions**

None.

**See Also**

dlerror(), dlopen(), dlsym(), the Base Definitions volume of IEEE Std 1003.1-2001, <dlfcn.h>

**Change History**

First released in Issue 5.

**Issue 6**

The DESCRIPTION is updated to say that the referenced object is closed “if this is the last reference to it”.
NAME
dlerror — get diagnostic information

SYNOPSIS
XSI
#include <dlfcn.h>
char *dlerror(void);

DESCRIPTION
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.
The dlerror() function need not be reentrant. A function that is not required to be reentrant is not
required to be thread-safe.

RETURN VALUE
If successful, dlerror() shall return a null-terminated character string; otherwise, NULL shall be
returned.

ERRORS
No errors are defined.

EXAMPLES
The following example prints out the last dynamic linking error:
...
#include <dlfcn.h>
char *errstr;
errstr = dlerror();
if (errstr != NULL)
printf("A dynamic linking error occurred: (%s)\n", errstr);
...

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
dlclose(), dlopen(), dlsym(), the Base Definitions volume of IEEE Std 1003.1-2001, <dlfcn.h>
CHANGE HISTORY
First released in Issue 5.

Issue 6
In the DESCRIPTION the note about reentrancy and thread-safety is added.
NAME
dlopen — gain access to an executable object file

SYNOPSIS
#include <dlfcn.h>

void *dlopen(const char *file, int mode);

DESCRIPTION
The dlopen() function shall make an executable object file specified by file available to the calling
program. The class of files eligible for this operation and the manner of their construction are
implementation-defined, though typically such files are executable objects such as shared
libraries, relocatable files, or programs. Note that some implementations permit the construction
of dependencies between such objects that are embedded within files. In such cases, a dlopen()
operation shall load such dependencies in addition to the object referenced by file. Implementations may also impose specific constraints on the construction of programs that can
employ dlopen() and its related services.

A successful dlopen() shall return a handle which the caller may use on subsequent calls to
dlsym() and dlclose(). The value of this handle should not be interpreted in any way by the caller.

The file argument is used to construct a pathname to the object file. If file contains a slash
character, the file argument is used as the pathname for the file. Otherwise, file is used in an
implementation-defined manner to yield a pathname.

If the value of file is 0, dlopen() shall provide a handle on a global symbol object. This object shall
provide access to the symbols from an ordered set of objects consisting of the original program
image file, together with any objects loaded at program start-up as specified by that process
image file (for example, shared libraries), and the set of objects loaded using a dlopen() operation
together with the RTLD_GLOBAL flag. As the latter set of objects can change during execution,
the set identified by handle can also change dynamically.

Only a single copy of an object file is brought into the address space, even if dlopen() is invoked
multiple times in reference to the file, and even if different pathnames are used to reference the
file.

The mode parameter describes how dlopen() shall operate upon file with respect to the processing
of relocations and the scope of visibility of the symbols provided within file. When an object is
brought into the address space of a process, it may contain references to symbols whose
addresses are not known until the object is loaded. These references shall be relocated before the
symbols can be accessed. The mode parameter governs when these relocations take place and
may have the following values:

RTLD_LAZY  Relocations shall be performed at an implementation-defined time,
           ranging from the time of the dlopen() call until the first reference to a
given symbol occurs. Specifying RTLD_LAZY should improve
           performance on implementations supporting dynamic symbol binding as
           a process may not reference all of the functions in any given object. And,
           for systems supporting dynamic symbol resolution for normal process
           execution, this behavior mimics the normal handling of process
           execution.

RTLD_NOW   All necessary relocations shall be performed when the object is first
           loaded. This may waste some processing if relocations are performed for
           functions that are never referenced. This behavior may be useful for
           applications that need to know as soon as an object is loaded that all
symbols referenced during execution are available.

Any object loaded by \textit{dlopen()} 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 \textit{dlopen()} invocation, and any objects that were loaded in any \textit{dlopen()} invocation and which specified the RTLD_GLOBAL flag. To determine the scope of visibility for the symbols loaded with a \textit{dlopen()} invocation, the \textit{mode} parameter should be a bitwise-inclusive OR with one of the following values:

- **RTLD_GLOBAL**: The object’s symbols shall be made available for the relocation processing of any other object. In addition, symbol lookup using \textit{dlopen(0, mode)} and an associated \textit{dlsym()} allows objects loaded with this mode to be searched.
- **RTLD_LOCAL**: The object’s symbols shall not be made available for the relocation processing of any other object.

If neither RTLD_GLOBAL nor RTLD_LOCAL are specified, then an implementation-defined default behavior shall be applied.

If a file is specified in multiple \textit{dlopen()} invocations, \textit{mode} 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 \textit{dlclose}()).

Symbols introduced into a program through calls to \textit{dlopen()} may be used in relocation activities. Symbols so introduced may duplicate symbols already defined by the program or previous \textit{dlopen()} 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: load or dependency 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 \textit{dlopen()}). 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 \textit{dlopen()} operation on a file of 0, dependency ordering is used by the \textit{dlsym()} function. Load ordering is used in \textit{dlsym}() operations upon the global symbol object.

When an object is first made accessible via \textit{dlopen()} 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.

The symbols introduced by \textit{dlopen()} operations and available through \textit{dlsym()} 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 \textit{extern} linkage. The precise manner in which an implementation constructs the set of exported symbols for a \textit{dlopen()} object is specified by that implementation.

**RETURN VALUE**

If file cannot be found, cannot be opened for reading, is not of an appropriate object format for processing by \textit{dlopen()}, or if an error occurs during the process of loading file or relocating its symbolic references, \textit{dlopen()} shall return NULL. More detailed diagnostic information shall be available through \textit{dlerror}().
dlopen()  

ERRORS  
No errors are defined.

EXAMPLES  
None.

APPLICATION USAGE  
None.

RATIONALE  
None.

FUTURE DIRECTIONS  
None.

SEE ALSO  

dlclose(), dlerror(), dlsym(), the Base Definitions volume of IEEE Std 1003.1-2001, <dlfcn.h>

CHANGE HISTORY  
First released in Issue 5.
NAME
dlsym — obtain the address of a symbol from a dlopen() object

SYNOPSIS
XSI
#include <dlfcn.h>

void *dlsym(void *restrict handle, const char *restrict name);

DESCRIPTION
The dlsym() function shall obtain the address of a symbol defined within an object made
accessible through a dlopen() call. The handle argument is the value returned from a call to
dlopen() (and which has not since been released via a call to dlclose()), and name is the symbol's
name as a character string.

The dlsym() function shall search for the named symbol in all objects loaded automatically as a
result of loading the object referenced by handle (see dlopen()). Load ordering is used in dlsym()
operations upon the global symbol object. The symbol resolution algorithm used shall be
dependency order as described in dlopen().

The RTLD_DEFAULT and RTLD_NEXT flags are reserved for future use.

RETURN VALUE
If handle does not refer to a valid object opened by dlopen(), or if the named symbol cannot be
found within any of the objects associated with handle, dlsym() shall return NULL. More
detailed diagnostic information shall be available through dlerror().

ERRORS
No errors are defined.

EXAMPLES
The following example shows how dlopen() and dlsym() can be used to access either function or
data objects. For simplicity, error checking has been omitted.

```c
void *handle;
int *iptr, (*fptr)(int);
/* open the needed object */
handle = dlopen("/usr/home/me/libfoo.so", RTLD_LOCAL | RTLD_LAZY);
/* find the address of function and data objects */
fptr = (int (*)(int))dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_object");
/* invoke function, passing value of integer as a parameter */
(*fptr)(*iptr);
```

APPLICATION USAGE
Special purpose values for handle are reserved for future use. These values and their meanings
are:

RTLD_DEFAULT The symbol lookup happens in the normal global scope; that is, a search for a
symbol using this handle would find the same definition as a direct use of this
symbol in the program code.

RTLD_NEXT Specifies the next object after this one that defines name. This one refers to the
object containing the invocation of dlsym(). The next object is the one found
upon the application of a load order symbol resolution algorithm (see dlopen()). The next object is either one of global scope (because it was
introduced as part of the original process image or because it was added with
The RTLD_GLOBAL flag, or is an object that was included in the same `dlopen()` operation that loaded this one.

The RTLD_NEXT flag is useful to navigate an intentionally created hierarchy of multiply-defined symbols created through interposition. For example, if a program wished to create an implementation of `malloc()` that embedded some statistics gathering about memory allocations, such an implementation could use the real `malloc()` definition to perform the memory allocation—and itself only embed the necessary logic to implement the statistics gathering function.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`dlsym()`, `dlerror()`, `dlopen()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<dlfcn.h>`

**CHANGE HISTORY**

First released in Issue 5.

**Issue 6**

The `restrict` keyword is added to the `dlsym()` prototype for alignment with the ISO/IEC 9899:1999 standard.

The RTLD_DEFAULT and RTLD_NEXT flags are reserved for future use.
NAME
drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, srand48 — generate
uniformly distributed pseudo-random numbers

SYNOPSIS
XSI
#include <stdlib.h>

double drand48(void);

double erand48(unsigned short xsubi[3]);

long jrand48(unsigned short xsubi[3]);

void lcong48(unsigned short param[7]);

long lrand48(void);

long mrand48(void);

long nrand48(unsigned short xsubi[3]);

unsigned short *seed48(unsigned short seed16v[3]);

void srand48(long seedval);

DESCRIPTION
This family of functions shall generate pseudo-random numbers using a linear congruential
algorithm and 48-bit integer arithmetic.

The drand48() and erand48() functions shall return non-negative, double-precision, floating-
point values, uniformly distributed over the interval [0.0,1.0).

The lrand48() and nrand48() functions shall return non-negative, long integers, uniformly
distributed over the interval [0,2^{31}).

The mrand48() and jrand48() functions shall return signed long integers uniformly distributed
over the interval [−2^{31},2^{31}).

The srand48(), seed48(), and lcong48() functions are initialization entry points, one of which
should be invoked before either drand48(), lrand48(), or mrand48() is called. (Although it is not
recommended practice, constant default initializer values shall be supplied automatically if
drand48(), lrand48(), or mrand48() is called without a prior call to an initialization entry point.)
The erand48(), nrand48(), and jrand48() functions do not require an initialization entry point to
be called first.

All the routines work by generating a sequence of 48-bit integer values, \( X_i \), according to the
linear congruential formula:

\[
X_{n+1} = (aX_n + c) \mod m \quad n \geq 0
\]

The parameter \( m = 2^{48} \); hence 48-bit integer arithmetic is performed. Unless lcong48() is invoked,
the multiplier value \( a \) and the addend value \( c \) are given by:

\[
a = 5DECE66D_{16} = 273673163155_8
\]

\[
c = B_{16} = 13_8
\]

The value returned by any of the drand48(), erand48(), jrand48(), lrand48(), mrand48(), or
nrand48() functions is computed by first generating the next 48-bit \( X_i \) in the sequence. Then the
appropriate number of bits, according to the type of data item to be returned, are copied from
the high-order (leftmost) bits of \( X_i \) and transformed into the returned value.

The drand48(), lrand48(), and mrand48() functions store the last 48-bit \( X_i \) generated in an
internal buffer; that is why the application shall ensure that these are initialized prior to being
invoked. The erand48(), nrand48(), and jrand48() functions require the calling program to
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, \( drand48() \), \( nrand48() \), 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 \( srand48() \) 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 \( 330E_{16} \).

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 \( srand48() \) or \( seed48() \) shall restore the standard multiplier and addend values, \( a \) and
\( c \), specified above.

The \( drand48() \), \( lrand48() \), and \( mrand48() \) functions need not be reentrant. A function that is not
required to be reentrant is not required to be thread-safe.

**RETURN VALUE**
As described in the DESCRIPTION above.

**ERRORS**
No errors are defined.

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
\( rand() \), the Base Definitions volume of IEEE Std 1003.1-2001, \(<stdlib.h>\)

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**
A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

**Issue 6**
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
dup, dup2 — duplicate an open file descriptor

SYNOPSIS
#include <unistd.h>
int dup(int fildes);
int dup2(int fildes, int fildes2);

DESCRIPTION
The dup() and dup2() functions provide an alternative interface to the service provided by
fcntl() using the F_DUPFD command. The call:

fid = dup(fildes);

shall be equivalent to:

fid = fcntl(fildes, F_DUPFD, 0);

The call:

fid = dup2(fildes, fildes2);

shall be equivalent to:

close(fildes2);

fid = fcntl(fildes, F_DUPFD, fildes2);

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.

RETURN VALUE
Upon successful completion a non-negative integer, namely the file descriptor, shall be returned;
otherwise, −1 shall be returned and errno set to indicate the error.

ERRORS
The dup() function shall fail if:

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

[EMFILE] The number of file descriptors in use by this process would exceed
{OPEN_MAX}.

The dup2() function shall fail if:

[EBADF] The fildes argument is not a valid open file descriptor or the argument fildes2 is
negative or greater than or equal to {OPEN_MAX}.

[EINTR] The dup2() function was interrupted by a signal.
EXAMPLES

Redirecting Standard Output to a File

The following example closes standard output for the current processes, re-assigns standard output to go to the file referenced by pfd, and closes the original file descriptor to clean up.

```c
#include <unistd.h>
...
int pfd;
...
close(1);
dup(pfd);
close(pfd);
...
```

Redirecting Error Messages

The following example redirects messages from stderr to stdout.

```c
#include <unistd.h>
...
dup2(1, 2);
...
```

APPLICATION USAGE

None.

RATIONALE

The `dup()` and `dup2()` functions are redundant. Their services are also provided by the `fcntl()` function. They have been included in this volume of IEEE Std 1003.1-2001 primarily for historical reasons, since many existing applications use them.

While the brief code segment shown is very similar in behavior to `dup2()`, a conforming implementation based on other functions defined in this volume of IEEE Std 1003.1-2001 is significantly more complex. Least obvious is the possible effect of a signal-catching function that could be invoked between steps and allocate or deallocate file descriptors. This could be avoided by blocking signals.

The `dup2()` function is not marked obsolescent because it presents a type-safe version of functionality provided in a type-unsafe version by `fcntl()`. It is used in the POSIX Ada binding.

The `dup2()` function is not intended for use in critical regions as a synchronization mechanism.

In the description of [EBADF], the case of `fildes` being out of range is covered by the given case of `fildes` not being valid. The descriptions for `fildes` and `fildes2` are different because the only kind of invalidity that is relevant for `fildes2` is whether it is out of range; that is, it does not matter whether `fildes2` refers to an open file when the `dup2()` call is made.

FUTURE DIRECTIONS

None.

SEE ALSO

`close()`, `fcntl()`, `open()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<unistd.h>`
CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
ecvt, fcvt, gcvt — convert a floating-point number to a string (LEGACY)

SYNOPSIS
#include <stdlib.h>

char *ecvt(double value, int ndigit, int *restrict decpt, int *restrict sign);
char *fcvt(double value, int ndigit, int *restrict decpt, int *restrict sign);
char *gcvt(double value, int ndigit, char *buf);

DESCRIPTION
The ecvt(), fcvt(), and gcvt() functions shall convert floating-point numbers to null-terminated strings.

The ecvt() function shall convert value to a null-terminated string of ndigit digits (where ndigit is reduced to an unspecified limit determined by the precision of a double) and return a pointer to the string. The high-order digit shall be non-zero, unless the value is 0. The low-order digit shall be rounded in an implementation-defined manner. The position of the radix character relative to the beginning of the string shall be stored in the integer pointed to by decpt (negative means to the left of the returned digits). If value is zero, it is unspecified whether the integer pointed to by decpt would be 0 or 1. The radix character shall not be included in the returned string. If the sign of the result is negative, the integer pointed to by sign shall be non-zero; otherwise, it shall be 0.

If the converted value is out of range or is not representable, the contents of the returned string are unspecified.

The fcvt() function shall be equivalent to ecvt(), except that ndigit specifies the number of digits desired after the radix character. The total number of digits in the result string is restricted to an unspecified limit as determined by the precision of a double.

The gcvt() function shall convert value to a null-terminated string (similar to that of the %g conversion specification format of printf()) in the array pointed to by buf and shall return buf. It shall produce ndigit significant digits (limited to an unspecified value determined by the precision of a double) in the %e conversion specification format of printf() if possible, or the %e conversion specification format of printf() (scientific notation) otherwise. A minus sign shall be included in the returned string if value is less than 0. A radix character shall be included in the returned string if value is not a whole number. Trailing zeros shall be suppressed where value is not a whole number. The radix character is determined by the current locale. If setlocale() has not been called successfully, the default locale, POSIX, is used. The default locale specifies a period (’. ’) as the radix character. The LC_NUMERIC category determines the value of the radix character within the current locale.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE
The ecvt() and fcvt() functions shall return a pointer to a null-terminated string of digits.

The gcvt() function shall return buf.

The return values from ecvt() and fcvt() may point to static data which may be overwritten by subsequent calls to these functions.
ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
The `sprintf()` function is preferred over this function.

RATIONALE
None.

FUTURE DIRECTIONS
These functions may be withdrawn in a future version.

SEE ALSO
`printf()`, `setlocale()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.
A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6
In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
This function is marked LEGACY.
The `restrict` keyword is added to the `ecvt()` and `fcvt()` prototypes for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION is updated to explicitly use “conversion specification” to describe `%g`, `%f`, and `%e.`
encrypt()  System Interfaces

NAME
encrypt — encoding function (CRYPT)

SYNOPSIS

```c
#include <unistd.h>

void encrypt(char block[64], int edflag);
```

DESCRIPTION
The `encrypt()` function shall provide access to an implementation-defined encoding algorithm. The key generated by `setkey()` is used to encrypt the string `block` with `encrypt()`.

The `block` argument to `encrypt()` shall be an array of length 64 bytes containing only the bytes with values of 0 and 1. The array is modified in place to a similar array using the key set by `setkey()`. If `edflag` is 0, the argument is encoded. If `edflag` is 1, the argument may be decoded (see the APPLICATION USAGE section); if the argument is not decoded, `errno` shall be set to [ENOSYS].

The `encrypt()` function shall not change the setting of `errno` if successful. An application wishing to check for error situations should set `errno` to 0 before calling `encrypt()`. If `errno` is non-zero on return, an error has occurred.

The `encrypt()` function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE
The `encrypt()` function shall not return a value.

ERRORS
The `encrypt()` function shall fail if:

- [ENOSYS] The functionality is not supported on this implementation.

EXAMPLES
None.

APPLICATION USAGE
Historical implementations of the `encrypt()` function used a rather primitive encoding algorithm. In some environments, decoding might not be implemented. This is related to some Government restrictions on encryption and decryption routines. Historical practice has been to ship a different version of the encryption library without the decryption feature in the routines supplied. Thus the exported version of `encrypt()` does encoding but not decoding.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
`crypt()`, `setkey()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<unistd.h>`

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

268 System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
A note indicating that this function need not be reentrant is added to the DESCRIPTION.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
NAME
endgrent, getgrent, setgrent — group database entry functions

SYNOPSIS
XSI
#include <grp.h>

void endgrent(void);
struct group *getgrent(void);
void setgrent(void);

DESCRIPTION
The getgrent() function shall return a pointer to a structure containing the broken-out fields of an entry in the group database. When first called, getgrent() shall return a pointer to a group structure containing the first entry in the group database. Thereafter, it shall return a pointer to a group structure containing the next group structure in the group database, so successive calls may be used to search the entire database.

An implementation that provides extended security controls may impose further implementation-defined restrictions on accessing the group database. In particular, the system may deny the existence of some or all of the group database entries associated with groups other than those groups associated with the caller and may omit users other than the caller from the list of members of groups in database entries that are returned.

The setgrent() function shall rewind the group database to allow repeated searches.

The endgrent() function may be called to close the group database when processing is complete.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE
When first called, getgrent() shall return a pointer to the first group structure in the group database. Upon subsequent calls it shall return the next group structure in the group database.

The getgrent() function shall return a null pointer on end-of-file or an error and errno may be set to indicate the error.

The return value may point to a static area which is overwritten by a subsequent call to getgrgid(), getgrnam(), or getgrent().

ERRORS
The getgrent() function may fail if:

EINVAL A signal was caught during the operation.

EIO An I/O error has occurred.

EMFILE (OPEN_MAX) file descriptors are currently open in the calling process.

ENFILE The maximum allowable number of files is currently open in the system.
EXAMPLES
None.

APPLICATION USAGE
These functions are provided due to their historical usage. Applications should avoid dependencies on fields in the group database, whether the database is a single file, or where in the file system name space the database resides. Applications should use getgrnam() and getgrgid() whenever possible because it avoids these dependencies.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getgrgid(), getgrnam(), getlogin(), getpwent(), the Base Definitions volume of IEEE Std 1003.1-2001, <grp.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
endhostent()

NAME
endhostent, gethostent, sethostent — network host database functions

SYNOPSIS
#include <netdb.h>
void endhostent(void);
struct hostent *gethostent(void);
void sethostent(int stayopen);

DESCRIPTION
These functions shall retrieve information about hosts. This information is considered to be
stored in a database that can be accessed sequentially or randomly. The implementation of this
database is unspecified.

Note: In many cases this database is implemented by the Domain Name System, as documented in
RFC 1034, RFC 1035, and RFC 1886.

The sethostent() function shall open a connection to the database and set the next entry for
retrieval to the first entry in the database. If the stayopen argument is non-zero, the connection
shall not be closed by a call to gethostent(), gethostbyname(), or gethostbyaddr(), and the
implementation may maintain an open file descriptor.

The gethostent() function shall read the next entry in the database, opening and closing a
connection to the database as necessary.

Entries shall be returned in hostent structures. Refer to gethostbyaddr() for a definition of the
hostent structure.

The endhostent() function shall close the connection to the database, releasing any open file
descriptor.

These functions need not be reentrant. A function that is not required to be reentrant is not
required to be thread-safe.

RETURN VALUE
Upon successful completion, the gethostent() function shall return a pointer to a hostent
structure if the requested entry was found, and a null pointer if the end of the database was
reached or the requested entry was not found.

ERRORS
No errors are defined for endhostent(), gethostent(), and sethostent().

EXAMPLES
None.

APPLICATION USAGE
The gethostent() function may return pointers to static data, which may be overwritten by
subsequent calls to any of these functions.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
endservent(), gethostbyaddr(), the Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>
CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
endnetent()

NAME
endnetent, getnetbyaddr, getnetbyname, getnetent, setnetent — network database functions

SYNOPSIS
#include <netdb.h>

#include <netdb.h>

void endnetent(void);
struct netent *getnetbyaddr(uint32_t net, int type);
struct netent *getnetbyname(const char *name);
struct netent *getnetent(void);
void setnetent(int stayopen);

DESCRIPTION
These functions shall retrieve information about networks. This information is considered to be
stored in a database that can be accessed sequentially or randomly. The implementation of this
database is unspecified.

The setnetent() function shall open and rewind the database. If the stayopen argument is non-
zero, the connection to the net database shall not be closed after each call to getnetent() (either
directly, or indirectly through one of the other getnet*() functions), and the implementation may
maintain an open file descriptor to the database.

The getnetent() function shall read the next entry of the database, opening and closing a
connection to the database as necessary.

The getnetbyaddr() function shall search the database from the beginning, and find the first entry
for which the address family specified by type matches the n_addrtype member and the network
number net matches the n_net member, opening and closing a connection to the database as
necessary. The net argument shall be the network number in host byte order.

The getnetbyname() function shall search the database from the beginning and find the first entry
for which the network name specified by name matches the n_name member, opening and
closing a connection to the database as necessary.

The getnetbyaddr(), getnetbyname(), and getnetent() functions shall each return a pointer to a
netent structure, the members of which shall contain the fields of an entry in the network
database.

The endnetent() function shall close the database, releasing any open file descriptor.

These functions need not be reentrant. A function that is not required to be reentrant is not
required to be thread-safe.

RETURN VALUE
Upon successful completion, getnetbyaddr(), getnetbyname(), and getnetent() shall return a
pointer to a netent structure if the requested entry was found, and a null pointer if the end of the
database was reached or the requested entry was not found. Otherwise, a null pointer shall be
returned.

ERRORS
No errors are defined.
EXAMPLES

None.

APPLICATION USAGE

The getnetbyaddr(), getnetbyname(), and getnetent() functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
endprotoent, getprotobyname, getprotobynumber, getprotoent, setprotoent — network protocol
database functions

SYNOPSIS
#include <netdb.h>

void endprotoent(void);
struct protoent *getprotobyname(const char *name);
struct protoent *getprotobynumber(int proto);
struct protoent *getprotoent(void);
void setprotoent(int stayopen);

DESCRIPTION
These functions shall retrieve information about protocols. This information is considered to be
stored in a database that can be accessed sequentially or randomly. The implementation of this
database is unspecified.

The setprotoent() function shall open a connection to the database, and set the next entry to the
first entry. If the stayopen argument is non-zero, the connection to the network protocol database
shall not be closed after each call to getprotoent() (either directly, or indirectly through one of the
other getproto*() functions), and the implementation may maintain an open file descriptor for
the database.

The getprotobyname() function shall search the database from the beginning and find the first
entry for which the protocol name specified by name matches the p_name member, opening and
closing a connection to the database as necessary.

The getprotobynumber() function shall search the database from the beginning and find the first
entry for which the protocol number specified by proto matches the p_proto member, opening
and closing a connection to the database as necessary.

The getprotoent() function shall read the next entry of the database, opening and closing a
connection to the database as necessary.

The getprotobyname(), getprotobynumber(), and getprotoent() functions shall each return a pointer
to a protoent structure, the members of which shall contain the fields of an entry in the network
protocol database.

The endprotoent() function shall close the connection to the database, releasing any open file
descriptor.

These functions need not be reentrant. A function that is not required to be reentrant is not
required to be thread-safe.

RETURN VALUE
Upon successful completion, getprotobyname(), getprotobynumber(), and getprotoent() return a
pointer to a protoent structure if the requested entry was found, and a null pointer if the end of
the database was reached or the requested entry was not found. Otherwise, a null pointer is
returned.

ERRORS
No errors are defined.
EXAMPLES
None.

APPLICATION USAGE
The getprotobyname(), getprotobynumber(), and getprotoent() functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
endpwent()

NAME
endpwent, getpwent, setpwent — user database functions

SYNOPSIS
XSI
#include <pwd.h>

void endpwent(void);
struct passwd *getpwent(void);
void setpwent(void);

DESCRIPTION
These functions shall retrieve information about users.
The getpwent() function shall return a pointer to a structure containing the broken-out fields of an entry in the user database. Each entry in the user database contains a passwd structure. When first called, getpwent() shall return a pointer to a passwd structure containing the first entry in the user database. Thereafter, it shall return a pointer to a passwd structure containing the next entry in the user database. Successive calls can be used to search the entire user database.

If an end-of-file or an error is encountered on reading, getpwent() shall return a null pointer.

An implementation that provides extended security controls may impose further implementation-defined restrictions on accessing the user database. In particular, the system may deny the existence of some or all of the user database entries associated with users other than the caller.

The setpwent() function effectively rewinds the user database to allow repeated searches.

The endpwent() function may be called to close the user database when processing is complete.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE
The getpwent() function shall return a null pointer on end-of-file or error.

ERRORS
The getpwent(), setpwent(), and endpwent() functions may fail if:

[EIO] An I/O error has occurred.

In addition, getpwent() and setpwent() may fail if:

[EMFILE] [OPEN_MAX] file descriptors are currently open in the calling process.

[ENFILE] The maximum allowable number of files is currently open in the system.

The return value may point to a static area which is overwritten by a subsequent call to getpwuid(), getpwnam(), or getpwent().
```c
#include <pwd.h>
...
struct passwd *p;
...
while ((p = getpwent ()) != NULL) {
    ...
}
endpwent();
...
```

**APPLICATION USAGE**

These functions are provided due to their historical usage. Applications should avoid dependencies on fields in the password database, whether the database is a single file, or where in the file system name space the database resides. Applications should use `getpwuid()` whenever possible because it avoids these dependencies.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`endgrent()`, `getlogin()`, `getpwnam()`, `getpwuid()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pwd.h>`

**CHANGE HISTORY**

First released in Issue 4, Version 2.

**Issue 5**

Moved from X/OPEN UNIX extension to BASE.

**Issue 6**

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
endservent()

NAME
endservent, getservbyname, getservbyport, getservent, setservent — network services database functions

SYNOPSIS
#include <netdb.h>

void endservent(void);

struct servent *getservbyname(const char *name, const char *proto);

struct servent *getservbyport(int port, const char *proto);

struct servent *getservent(void);

void setservent(int stayopen);

DESCRIPTION
These functions shall retrieve information about network services. This information is considered to be stored in a database that can be accessed sequentially or randomly. The implementation of this database is unspecified.

The setservent() function shall open a connection to the database, and set the next entry to the first entry. If the stayopen argument is non-zero, the net database shall not be closed after each call to the getservent() function (either directly, or indirectly through one of the other getsero*( ) functions), and the implementation may maintain an open file descriptor for the database.

The getservent() function shall read the next entry of the database, opening and closing a connection to the database as necessary.

The getservbyname() function shall search the database from the beginning and find the first entry for which the service name specified by name matches the s_name member and the protocol name specified by proto matches the s_proto member, opening and closing a connection to the database as necessary. If proto is a null pointer, any value of the s_proto member shall be matched.

The getservbyport() function shall search the database from the beginning and find the first entry for which the port specified by port matches the s_port member and the protocol name specified by proto matches the s_proto member, opening and closing a connection to the database as necessary. If proto is a null pointer, any value of the s_proto member shall be matched. The port argument shall be in network byte order.

The getservbyname(), getservbyport(), and getservent() functions shall each return a pointer to a servent structure, the members of which shall contain the fields of an entry in the network services database.

The endservent() function shall close the database, releasing any open file descriptor.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE
Upon successful completion, getservbyname(), getservbyport(), and getservent() return a pointer to a servent structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.

ERRORS
No errors are defined.
EXAMPLES
None.

APPLICATION USAGE
The port argument of getservbyport() need not be compatible with the port values of all address families.

The getservbyname(), getservbyport(), and getservent() functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
endhostent(), endprotoent(), htonl(), inet_addr(), the Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
endutxent, getutxent, getutxid, getutxline, pututxline, setutxent — user accounting database functions

SYNOPSIS
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);

DESCRIPTION
These functions shall provide access to the user accounting database.

The getutxent() function shall read the next entry from the user accounting database. If the database is not already open, it shall open it. If it reaches the end of the database, it shall fail.

The getutxid() function shall search forward from the current point in the database. If the ut_type value of the utmpx structure pointed to by id is BOOT_TIME, OLD_TIME, or NEW_TIME, then it shall stop when it finds an entry with a matching ut_type value. If the ut_type value is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or DEAD_PROCESS, then it shall stop when it finds an entry whose type is one of these four and whose ut_id member matches the ut_id member of the utmpx structure pointed to by id. If the end of the database is reached without a match, getutxid() shall fail.

The getutxline() function shall search forward from the current point in the database until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a ut_line value matching that in the utmpx structure pointed to by line. If the end of the database is reached without a match, getutxline() shall fail.

The getutxid() or getutxline() function may cache data. For this reason, to use getutxline() to search for multiple occurrences, the application shall zero out the static data after each success, or getutxline() may return a pointer to the same utmpx structure.

There is one exception to the rule about clearing the structure before further reads are done. The implicit read done by pututxline() (if it finds that it is not already at the correct place in the user accounting database) shall not modify the static structure returned by getutxent(), getutxid(), or getutxline(), if the application has modified this structure and passed the pointer back to pututxline().

For all entries that match a request, the ut_type member indicates the type of the entry. Other members of the entry shall contain meaningful data based on the value of the ut_type member as follows:
### System Interfaces

#### endutxent()

<table>
<thead>
<tr>
<th>ut_type Member</th>
<th>Other Members with Meaningful Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPTY</td>
<td>No others</td>
</tr>
<tr>
<td>BOOT_TIME</td>
<td>ut_tv</td>
</tr>
<tr>
<td>OLD_TIME</td>
<td>ut_tv</td>
</tr>
<tr>
<td>NEW_TIME</td>
<td>ut_tv</td>
</tr>
<tr>
<td>USER_PROCESS</td>
<td>ut_id, ut_user (login name of the user), ut_line, ut_pid, ut_tv</td>
</tr>
<tr>
<td>INIT_PROCESS</td>
<td>ut_id, ut_pid, ut_tv</td>
</tr>
<tr>
<td>LOGIN_PROCESS</td>
<td>ut_id, ut_user (implementation-defined name of the login process), ut_pid, ut_tv</td>
</tr>
<tr>
<td>DEAD_PROCESS</td>
<td>ut_id, ut_pid, ut_tv</td>
</tr>
</tbody>
</table>

An implementation that provides extended security controls may impose implementation-defined restrictions on accessing the user accounting database. In particular, the system may deny the existence of some or all of the user accounting database entries associated with users other than the caller.

If the process has appropriate privileges, the `pututxline()` function shall write out the structure into the user accounting database. It shall use `getutxid()` to search for a record that satisfies the request. If this search succeeds, then the entry shall be replaced. Otherwise, a new entry shall be made at the end of the user accounting database.

The `endutxent()` function shall close the user accounting database.

The `setutxent()` function shall reset the input to the beginning of the database. This should be done before each search for a new entry if it is desired that the entire database be examined.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

### RETURN VALUE

Upon successful completion, `getutxent()`, `getutxid()`, and `getutxline()` shall return a pointer to a `utmpx` structure containing a copy of the requested entry in the user accounting database. Otherwise, a null pointer shall be returned.

The return value may point to a static area which is overwritten by a subsequent call to `getutxid()` or `getutxline()`.

Upon successful completion, `pututxline()` shall return a pointer to a `utmpx` structure containing a copy of the entry added to the user accounting database. Otherwise, a null pointer shall be returned.

The `endutxent()` and `setutxent()` functions shall not return a value.

### ERRORS

No errors are defined for the `endutxent()`, `getutxent()`, `getutxid()`, `getutxline()`, and `setutxent()` functions.

The `pututxline()` function may fail if:

- [EPERM] The process does not have appropriate privileges.
endutxent()

EXCEPTIONS
None.

APPLICATION USAGE
The sizes of the arrays in the structure can be found using the sizeof operator.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <utmpx.h>

DESCRIPTION
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.
A note indicating that these functions need not be reentrant is added to the DESCRIPTION.
In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

CHANGE HISTORY
Moved from X/OPEN UNIX extension to BASE.
Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.
A note indicating that these functions need not be reentrant is added to the DESCRIPTION.
In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
NAME
environ — array of character pointers to the environment strings

SYNOPSIS
extern char **environ;

DESCRIPTION
Refer to the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables and exec.
erand48() — generate uniformly distributed pseudo-random numbers

SYNOPSIS

```c
#include <stdlib.h>

double erand48(unsigned short xsubi[3]);
```

DESCRIPTION

Refer to `drand48()`.
NAME
erf, erff, erfl — error functions

SYNOPSIS
#include <math.h>
double erf(double x);
float erff(float x);
long double erfl(long double x);

DESCRIPTION
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
These functions shall compute the error function of their argument \( x \), defined as:
\[
\frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} dt
\]
An application wishing to check for error situations should set \textit{errno} to zero and call
\textit{feclearexcept} (FE_ALL_EXCEPT) before calling these functions. On return, if \textit{errno} is non-zero or
\textit{fetestexcept} (FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE
Upon successful completion, these functions shall return the value of the error function.

If \( x \) is NaN, a NaN shall be returned.
If \( x \) is \( \pm 0 \), \( \pm 0 \) shall be returned.
If \( x \) is \( \pm \infty \), \( \pm 1 \) shall be returned.
If \( x \) is subnormal, a range error may occur, and \( 2 * x / \sqrt{\pi} \) should be returned.

ERRORS
These functions may fail if:

Range Error The result underflows.
If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
then \textit{errno} shall be set to \textit{[ERANGE]}. If the integer expression
(math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
floating-point exception shall be raised.

EXAMPLES
None.

APPLICATION USAGE
Underflow occurs when \( |x| < DBL_MIN * (\sqrt{\pi}/2) \).
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.

RATIONALE
None.
FUTURE DIRECTIONS

None.

SEE ALSO

erfc(), 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>

CHANGE HISTORY

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

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The erf() function is no longer marked as an extension.
The erfc() function is now split out onto its own reference page.
The erff() and erfl() functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.
NAME
erfc, erfcf, erfcl — complementary error functions

SYNOPSIS
#include <math.h>

double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);

DESCRIPTION
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

These functions shall compute the complementary error function 1.0 − erf(x).

An application wishing to check for error situations should set errno to zero and call
fecakeexcept(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.

RETURN VALUE
Upon successful completion, these functions shall return the value of the complementary error
function.

If the correct value would cause underflow and is not representable, a range error may occur
and either 0.0 (if representable), or an implementation-defined value shall be returned.

If x is NaN, a NaN shall be returned.

If x is ±0, +1 shall be returned.

If x is −Inf, +2 shall be returned.

If x is +Inf, +0 shall be returned.

If the correct value would cause underflow and is representable, a range error may occur and the
correct value shall be returned.

ERRORS
These functions may fail if:

Range Error The result underflows.

If the integer expression (math_errno & MATH_ERRNO) is non-zero,
then errno shall be set to [ERANGE]. If the integer expression
(math_errno & MATH_ERREXCEPT) is non-zero, then the underflow
floating-point exception shall be raised.

APPLICATION USAGE
The erfc() function is provided because of the extreme loss of relative accuracy if erf(x) is called
for large x and the result subtracted from 1.0.

Note for IEEE Std 754-1985 double, 26.55 < x implies erfc(x) has underflowed.

On error, the expressions (math_errno & MATH_ERRNO) and (math_errno &
MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.
RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
erf(), 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>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The erf() function is no longer marked as an extension.
These functions are split out from the erf() reference page.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899: 1999 standard.
NAME
erff, erfl — error functions

SYNOPSIS
#include <math.h>

float erff(float x);
long double erfl(long double x);

DESCRIPTION
Refer to erf().
The value of \texttt{errno} shall be defined only after a call to a function for which it is explicitly stated to be set and until it is changed by the next function call or if the application assigns it a value. The value of \texttt{errno} should only be examined when it is indicated to be valid by a function’s return value. Applications shall obtain the definition of \texttt{errno} by the inclusion of \texttt{<errno.h>}. No function in this volume of IEEE Std 1003.1-2001 shall set \texttt{errno} to 0.

It is unspecified whether \texttt{errno} is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name \texttt{errno}, the behavior is undefined.

The symbolic values stored in \texttt{errno} are documented in the ERRORS sections on all relevant pages.

\textbf{RETURN VALUE}

None.

\textbf{ERRORS}

None.

\textbf{EXAMPLES}

None.

\textbf{APPLICATION USAGE}

Previously both POSIX and X/Open documents were more restrictive than the ISO C standard in that they required \texttt{errno} to be defined as an external variable, whereas the ISO C standard required only that \texttt{errno} be defined as a modifiable lvalue with type \texttt{int}.

An application that needs to examine the value of \texttt{errno} to determine the error should set it to 0 before a function call, then inspect it before a subsequent function call.

\textbf{RATIONALE}

None.

\textbf{FUTURE DIRECTIONS}

None.

\textbf{SEE ALSO}

Section 2.3, the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<errno.h>}

\textbf{CHANGE HISTORY}

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

Issue 5

The following sentence is deleted from the DESCRIPTION: ‘The value of \texttt{errno} is 0 at program start-up, but is never set to 0 by any XSI function’. The DESCRIPTION also no longer states that conforming implementations may support the declaration:

\begin{verbatim}
extern int errno;
\end{verbatim}
Issue 6

Obsolescent text regarding defining \texttt{errno} as:

\begin{verbatim}
extern int errno
\end{verbatim}

is removed.

Text regarding no function setting \texttt{errno} to zero to indicate an error is changed to no function shall set \texttt{errno} to zero. This is for alignment with the ISO/IEC 9899:1999 standard.
NAME
environ, execl, execv, execl, execve, execlp, execvp — execute a file

SYNOPSIS
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ... /*,
(char *)0, char *const envp[]*/);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *const argv[]);

DESCRIPTION
The exec family of functions shall replace the current process image with a new process image.
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.

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:

int main (int argc, char *argv[]);

where argc is the argument count and argv is an array of character pointers to the arguments
themselves. In addition, the following variable:

extern char **environ;

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.

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.

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.

The argument path points to a pathname that identifies the new process image file.

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.

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 execlp() 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 execl() as follows:
exec(\texttt{<shell path>}, \texttt{arg0, file, arg1, ...}, (char *)0);

where \texttt{<shell path>} is an unspecified pathname for the \texttt{sh} utility, \texttt{file} is the process image file, and for \texttt{execvp()}, where \texttt{arg0, arg1, and so on} correspond to the values passed to \texttt{execvp()} in \texttt{argv[0], argv[1], and so on}.

The arguments represented by \texttt{arg0, ...} are pointers to null-terminated character strings. These strings shall constitute the argument list available to the new process image. The list is terminated by a null pointer. The argument \texttt{argv0} should point to a filename that is associated with the process being started by one of the \texttt{exec} functions.

The argument \texttt{argv} is an array of character pointers to null-terminated strings. The application shall ensure that the last member of this array is a null pointer. These strings shall constitute the argument list available to the new process image. The value in \texttt{argv[0]} should point to a filename that is associated with the process being started by one of the \texttt{exec} functions.

The argument \texttt{envp} is an array of character pointers to null-terminated strings. These strings shall constitute the environment for the new process image. The \texttt{envp} array is terminated by a null pointer.

For those forms not containing an \texttt{envp} pointer (\texttt{exel()}, \texttt{exec()}, \texttt{execvp()}, and \texttt{execvp()}), the environment for the new process image shall be taken from the external variable \texttt{environ} in the calling process.

The number of bytes available for the new process’ combined argument and environment lists is \{ARG_MAX\}. It is implementation-defined whether null terminators, pointers, and/or any alignment bytes are included in this total.

File descriptors open in the calling process image shall remain open in the new process image, except for those whose close-on-exec flag \texttt{FD_CLOEXEC} is set. For those file descriptors that remain open, all attributes of the open file description remain unchanged. For any file descriptor that is closed for this reason, file locks are removed as a result of the close as described in \texttt{close()}. Locks that are not removed by closing of file descriptors remain unchanged.

Directory streams open in the calling process image shall be closed in the new process image.

The state of the floating-point environment in the new process image shall be set to the default.

XSI The state of conversion descriptors and message catalog descriptors in the new process image is undefined. For the new process image, the equivalent of:

\begin{verbatim}
setlocale(LC_ALL, "C")
\end{verbatim}

shall be executed at start-up.

Signals set to the default action (SIG_DFL) in the calling process image shall be set to the default action in the new process image. Except for SIGCHLD, signals set to be ignored (SIG_IGN) by the calling process image shall be set to be ignored by the new process image. Signals set to be caught by the calling process image shall be set to the default action in the new process image (see \texttt{<signal.h>}). If the SIGCHLD signal is set to be ignored by the calling process image, it is unspecified whether the SIGCHLD signal is set to be ignored or to the default action in the new process image. After a successful call to any of the \texttt{exec} functions, alternate signal stacks are not preserved and the \texttt{SA_ONSTACK} flag shall be cleared for all signals.

After a successful call to any of the \texttt{exec} functions, any functions previously registered by \texttt{atexit()} are no longer registered.

XSI If the ST_NOSUID 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 `setuid()`.

XSI Any shared memory segments attached to the calling process image shall not be attached to the new process image.

SEMs Any named semaphores open in the calling process shall be closed as if by appropriate calls to `sem_close()`.

TMs Any blocks of typed memory that were mapped in the calling process are unmapped, as if `munmap()` was implicitly called to unmap them.

ML Memory locks established by the calling 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 the exec function. If the exec function fails, the effect on memory locks is unspecified.

SHM Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image.

PS For the SCHED_FIFO and SCHED_RR scheduling policies, the policy and priority settings shall not be changed by a call to an exec function. For other scheduling policies, the policy and priority settings on exec are implementation-defined.

TMR Per-process timers created by the calling process shall be deleted before replacing the current process image with the new process image.

MSG All open message queue descriptors in the calling process shall be closed, as described in `mq_close()`.

AIO Any outstanding asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled shall complete as if the exec function had not yet occurred, but any associated signal notifications shall be suppressed. It is unspecified whether the exec function itself blocks awaiting such I/O completion. In no event, however, shall the new process image created by the exec function be affected by the presence of outstanding asynchronous I/O operations at the time the exec function is called. Whether any I/O is canceled, and which I/O may be canceled upon exec, is implementation-defined.

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.

TCT The initial value of the CPU-time clock of the initial thread of the new process image shall be set to zero.

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 `posix_trace_eventid_open()` or the `posix_trace_trid_eventid_open()` functions in the calling process image.
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 `posix_trace_shutdown()` function.

The new process shall inherit at least the following attributes from the calling process image:

- Nice value (see `nice()`)
- `semadj` values (see `semop()`)
- Process ID
- Parent process ID
- Process group ID
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- Time left until an alarm clock signal (see `alarm()`)
- Current working directory
- Root directory
- File mode creation mask (see `umask()`)
- File size limit (see `ulimit()`)
- Process signal mask (see `sigprocmask()`)
- Pending signal (see `sigpending()`)
- `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` (see `times()`)
- Resource limits
- Controlling terminal
- Interval timers

All other process attributes defined in this volume of IEEE Std 1003.1-2001 shall be the same in the new and old process images. The inheritance of process attributes not defined by this volume of IEEE Std 1003.1-2001 is implementation-defined.

A call to any `exec` 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 shall be called.

Upon successful completion, the `exec` functions shall mark for update the `st_atime` field of the file. If an `exec` function failed but was able to locate the process image file, whether the `st_atime` field is marked for update is unspecified. Should the `exec` function succeed, the process image file shall be considered to have been opened with `open()`. The corresponding `close()` 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 `exec` functions, `posix_spawn()`, or `posix_spawnp()`. The `argv[]` and `envp[]` arrays of pointers and the strings to which those arrays point shall not be modified by a call to one of the `exec` functions, except as a consequence of replacing the process image.

The saved resource limits in the new process image are set to be a copy of the process’ corresponding hard and soft limits.
RETURN VALUE

If one of the exec functions returns to the calling process image, an error has occurred; the return value shall be −1, and errno shall be set to indicate the error.

ERRORS

The exec functions shall fail if:

- [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.
- [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.
- [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.
- [ELOOP] A loop exists in symbolic links encountered during resolution of the path or file argument.
- [ENAMETOOLONG] The length of the path or file arguments exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
- [ENOENT] A component of path or file does not name an existing file or path or file is an empty string.
- [ENOTDIR] A component of the new process image file’s path prefix is not a directory.

The exec functions, except for execvp() and execlp(), shall fail if:

- [ENOEXEC] The new process image file has the appropriate access permission but has an unrecognized format.

The exec functions may fail if:

- [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the path or file argument.
- [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}.
- [ENOMEM] The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.
- [ETXTBSY] The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.
EXAMPLES

Using `exec()`

The following example executes the `ls` command, specifying the pathname of the executable (`/bin/ls`) and using arguments supplied directly to the command to produce single-column output.

```c
#include <unistd.h>
int ret;
...
ret = execl ("/bin/ls", "ls", ",-1", (char *)0);
```

Using `execle()`

The following example is similar to Using `execl()`. In addition, it specifies the environment for the new process image using the `env` argument.

```c
#include <unistd.h>
int ret;
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
...
ret = execle ("/bin/ls", "ls", "-1", (char *)0, env);
```

Using `execlp()`

The following example searches for the location of the `ls` command among the directories specified by the `PATH` environment variable.

```c
#include <unistd.h>
int ret;
...
ret = execlp ("ls", "ls", ",-1", (char *)0);
```

Using `execv()`

The following example passes arguments to the `ls` command in the `cmd` array.

```c
#include <unistd.h>
int ret;
char *cmd[] = { "ls", ",-1", (char *)0 };
...
ret = execv ("/bin/ls", cmd);
```
Using execve()

The following example passes arguments to the `ls` command in the `cmd` array, and specifies the environment for the new process image using the `env` argument.

```c
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };  
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };  
...  
ret = execve ("/bin/ls", cmd, env);
```

Using execvp()

The following example searches for the location of the `ls` command among the directories specified by the `PATH` environment variable, and passes arguments to the `ls` command in the `cmd` array.

```c
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };  
...  
ret = execvp ("ls", cmd);
```

APPLICATION USAGE

As the state of conversion descriptors and message catalog descriptors in the new process image is undefined, conforming applications should not rely on their use and should close them prior to calling one of the exec functions.

Applications that require other than the default POSIX locale should call `setlocale()` with the appropriate parameters to establish the locale of the new process.

The `environ` array should not be accessed directly by the application.

RATIONALE

Early proposals required that the value of `argc` passed to `main()` be “one or greater”. This was driven by the same requirement in drafts of the ISO C standard. In fact, historical implementations have passed a value of zero when no arguments are supplied to the caller of the exec functions. This requirement was removed from the ISO C standard and subsequently removed from this volume of IEEE Std 1003.1-2001 as well. The wording, in particular the use of the word should, requires a Strictly Conforming POSIX Application to pass at least one argument to the exec function, thus guaranteeing that `argc` be one or greater when invoked by such an application. In fact, this is good practice, since many existing applications reference `argv[0]` without first checking the value of `argc`.

The requirement on a Strictly Conforming POSIX Application also states that the value passed as the first argument be a filename associated with the process being started. Although some existing applications pass a pathname rather than a filename in some circumstances, a filename is more generally useful, since the common usage of `argv[0]` is in printing diagnostics. In some cases the filename passed is not the actual filename of the file; for example, many implementations of the `login` utility use a convention of prefixing a hyphen (`'-'`) to the actual filename, which indicates to the command interpreter being invoked that it is a `‘login shell’`.

Historically there have been two ways that implementations can exec shell scripts.
One common historical implementation is that the `exec()` , `execv()` , `execle()` , and `execve()` functions return an [ENOEXEC] error for any file not recognizable as executable, including a shell script. When the `execp()` and `execvp()` functions encounter such a file, they assume the file to be a shell script and invoke a known command interpreter to interpret such files. This is now required by IEEE Std 1003.1-2001. These implementations of `execvp()` and `execlp()` only give the [ENOEXEC] error in the rare case of a problem with the command interpreter’s executable file. Because of these implementations, the [ENOEXEC] error is not mentioned for `execlp()` or `execvp()` , although implementations can still give it.

Another way that some historical implementations handle shell scripts is by recognizing the first two bytes of the file as the character string "#!" and using the remainder of the first line of the file as the name of the command interpreter to execute.

One potential source of confusion noted by the standard developers is over how the contents of a process image file affect the behavior of the exec family of functions. The following is a description of the actions taken:

1. If the process image file is a valid executable (in a format that is executable and valid and having appropriate permission) for this system, then the system executes the file.
2. If the process image file has appropriate permission and is in a format that is executable but not valid for this system (such as a recognized binary for another architecture), then this is an error and `errno` is set to [EINVAL] (see later RATIONALE on [EINVAL]).
3. If the process image file has appropriate permission but is not otherwise recognized:
   a. If this is a call to `execlp()` or `execvp()` , then they invoke a command interpreter assuming that the process image file is a shell script.
   b. If this is not a call to `execlp()` or `execvp()` , then an error occurs and `errno` is set to [ENOEXEC].

Applications that do not require to access their arguments may use the form:

```c
main(void)
```

as specified in the ISO C standard. However, the implementation will always provide the two arguments `argc` and `argv` , even if they are not used.

Some implementations provide a third argument to `main()` called `envp` . This is defined as a pointer to the environment. The ISO C standard specifies invoking `main()` with two arguments, so implementations must support applications written this way. Since this volume of IEEE Std 1003.1-2001 defines the global variable `environ` , which is also provided by historical implementations and can be used anywhere that `envp` could be used, there is no functional need for the `envp` argument. Applications should use the `getenv()` function rather than accessing the environment directly via either `envp` or `environ` . Implementations are required to support the two-argument calling sequence, but this does not prohibit an implementation from supporting `envp` as an optional third argument.

This volume of IEEE Std 1003.1-2001 specifies that signals set to SIG_IGN remain set to SIG_IGN, and that the process signal mask be unchanged across an exec. This is consistent with historical implementations, and it permits some useful functionality, such as the `nohup` command. However, it should be noted that many existing applications wrongly assume that they start with certain signals set to the default action and/or unblocked. In particular, applications written with a simpler signal model that does not include blocking of signals, such as the one in the ISO C standard, may not behave properly if invoked with some signals blocked. Therefore, it is best not to block or ignore signals across execs without explicit reason to do so, and especially not to block signals across execs of arbitrary (not closely co-operating) programs.
The `exec` functions always save the value of the effective user ID and effective group ID of the process at the completion of the `exec`, whether or not the set-user-ID or the set-group-ID bit of the process image file is set.

The statement about `argv[]` and `envp[]` being constants is included to make explicit to future writers of language bindings that these objects are completely constant. Due to a limitation of the ISO C standard, it is not possible to state that idea in standard C. Specifying two levels of `const-qualification` for the `argv[]` and `envp[]` parameters for the `exec` functions may seem to be the natural choice, given that these functions do not modify either the array of pointers or the characters to which the function points, but this would disallow existing correct code. Instead, only the array of pointers is noted as constant. The table of assignment compatibility for `dst=src` derived from the ISO C standard summarizes the compatibility:

<table>
<thead>
<tr>
<th><code>dst:</code></th>
<th><code>char *[]{  }</code></th>
<th><code>const char *[]{  }</code></th>
<th><code>char *const[]{  }</code></th>
<th><code>const char *const[]{  }</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>src:</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>char *[]{  }</code></td>
<td>VALID</td>
<td>—</td>
<td>VALID</td>
<td>—</td>
</tr>
<tr>
<td><code>const char *[]{  }</code></td>
<td>—</td>
<td>VALID</td>
<td>—</td>
<td>VALID</td>
</tr>
<tr>
<td><code>char * const[]{  }</code></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><code>const char *const[]{  }</code></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>VALID</td>
</tr>
</tbody>
</table>

Since all existing code has a source type matching the first row, the column that gives the most valid combinations is the third column. The only other possibility is the fourth column, but using it would require a cast on the `argv` or `envp` arguments. It is unfortunate that the fourth column cannot be used, because the declaration a non-expert would naturally use would be that in the second row.

The ISO C standard and this volume of IEEE Std 1003.1-2001 do not conflict on the use of `environ`, but some historical implementations of `environ` may cause a conflict. As long as `environ` is treated in the same way as an entry point (for example, `fork()`), it conforms to both standards. A library can contain `fork()`, but if there is a user-provided `fork()`, that `fork()` is given precedence and no problem ensues. The situation is similar for `environ`: the definition in this volume of IEEE Std 1003.1-2001 is to be used if there is no user-provided `environ` to take precedence. At least three implementations are known to exist that solve this problem.

- `[E2BIG]` The limit `{ARG_MAX}` applies not just to the size of the argument list, but to the sum of that and the size of the environment list.
- `[EFAULT]` Some historical systems return `[EFAULT]` rather than `[ENOEXEC]` when the new process image file is corrupted. They are non-conforming.
- `[EINVAL]` This error condition was added to IEEE Std 1003.1-2001 to allow an implementation to detect executable files generated for different architectures, and indicate this situation to the application. Historical implementations of shells, `execvp()`, and `execlp()` that encounter an `[ENOEXEC]` error will execute a shell on the assumption that the file is a shell script. This will not produce the desired effect when the file is a valid executable for a different architecture. An implementation may now choose to avoid this problem by returning `[EINVAL]` when a valid executable for a different architecture is encountered. Some historical implementations return `[EINVAL]` to indicate that the `path` argument contains a character with the high order bit set. The standard developers chose to deviate from historical practice for the following reasons:
  1. The new utilization of `[EINVAL]` will provide some measure of utility to the user community.
2. Historical use of [EINVAL] is not acceptable in an internationalized operating environment.

[ENAMETOOLONG] Since the file pathname may be constructed by taking elements in the PATH variable and putting them together with the filename, the [ENAMETOOLONG] error condition could also be reached this way.

[ETXTBSY] System V returns this error when the executable file is currently open for writing by some process. This volume of IEEE Std 1003.1-2001 neither requires nor prohibits this behavior.

Other systems (such as System V) may return [EINTR] from exec. This is not addressed by this volume of IEEE Std 1003.1-2001, but implementations may have a window between the call to exec and the time that a signal could cause one of the exec calls to return with [EINTR].

An explicit statement regarding the floating-point environment (as defined in the <fenv.h> header) was added to make it clear that the floating-point environment is set to its default when a call to one of the exec functions succeeds. The requirements for inheritance or setting to the default for other process and thread start-up functions is covered by more generic statements in their descriptions and can be summarized as follows:

posix_spawn() Set to default.
fork() Inherit.
pthread_create() Inherit.

FUTURE DIRECTIONS
None.

SEE ALSO
alarm(), atexit(), chmod(), close(), exit(), fcntl(), fork(), fstatvfs(), getenv(), getitimer(), getrlimit(), mmap(), nice(), posix_spawn(), posix_trace_eventid_open(), posix_trace_shutdown(), posix_trace_trid_eventid_open(), putenv(), semop(), setlocale(), shmat(), sigaction(), sigaltstack(), sigpending(), sigprocmask(), system(), times(), ulimit(), umask(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <unistd.h>

CHANGE HISTORY

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

Issue 5
The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

Issue 6
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, behavior is defined for when the process image file is not a valid executable.

- In this issue, _POSIX_SAVED_IDS is mandated, thus the effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by the setuid() function.

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

• The [ETXTBSY] optional error condition is added.

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

• The [EINVAL] mandatory error condition is added.

• The [ELOOP] optional error condition is added.

The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for typed memory.

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

The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #132 is applied.

The DESCRIPTION is updated to make it explicit that the floating-point environment in the new process image is set to the default.

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.
NAME
exit, _Exit, _exit — terminate a process

SYNOPSIS
#include <stdlib.h>
void exit(int status);
void _Exit(int status);
#include <unistd.h>
void _exit(int status);

DESCRIPTION
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.

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.

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.

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.

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.

The _Exit() and _exit() functions shall be functionally equivalent.

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.

These functions shall terminate the calling process with the following consequences:

Note: These consequences are all extensions to the ISO C standard and are not further CX shaded. However, XSI extensions are shaded.

• All of the file descriptors, directory streams, conversion descriptors, and message catalog descriptors open in the calling process shall be closed.

• 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 are 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().

The semantics of the waitid() function shall be equivalent to wait().

• 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
exit()

deleted at some later time when its parent process executes wait() or waitpid().

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:

If the implementation supports the SIGCHLD signal, a SIGCHLD shall be sent to the parent process.

Or:

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.
The _Exit() or _exit() operation may block awaiting such I/O completion. Whether any I/O is canceled, and which I/O may be canceled upon _Exit() or _exit(), is implementation-defined.

Threads terminated by a call to _Exit() or _exit() shall not invoke their cancelation cleanup handlers or per-thread data destructors.

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 by the posix_trace_shutdown() function, and any process’ mapping of trace event names to trace event type identifiers built for these trace streams may be deallocated.

**RETURN VALUE**
These functions do not return.

**ERRORS**
No errors are defined.

**EXAMPLES**
None.

**APPLICATION USAGE**
Normally applications should use exit() rather than _Exit() or _exit().

**RATIONALE**

**Process Termination**

Early proposals drew a distinction between normal and abnormal process termination. Abnormal termination was caused only by certain signals and resulted in implementation-defined “actions”, as discussed below. Subsequent proposals distinguished three types of termination: normal termination (as in the current specification), simple abnormal termination, and abnormal termination with actions. Again the distinction between the two types of abnormal termination was that they were caused by different signals and that implementation-defined actions would result in the latter case. Given that these actions were completely implementation-defined, the early proposals were only saying when the actions could occur and how their occurrence could be detected, but not what they were. This was of little or no use to conforming applications, and thus the distinction is not made in this volume of IEEE Std 1003.1-2001.

The implementation-defined actions usually include, in most historical implementations, the creation of a file named core in the current working directory of the process. This file contains an image of the memory of the process, together with descriptive information about the process, perhaps sufficient to reconstruct the state of the process at the receipt of the signal.

There is a potential security problem in creating a core file if the process was set-user-ID and the current user is not the owner of the program, if the process was set-group-ID and none of the user’s groups match the group of the program, or if the user does not have permission to write in the current directory. In this situation, an implementation either should not create a core file or should make it unreadable by the user.

Despite the silence of this volume of IEEE Std 1003.1-2001 on this feature, applications are advised not to create files named core because of potential conflicts in many implementations. Some implementations use a name other than core for the file; for example, by appending the process ID to the filename.
Terminating a Process

It is important that the consequences of process termination as described occur regardless of whether the process called \_exit() (perhaps indirectly through exit()) or instead was terminated due to a signal or for some other reason. Note that in the specific case of exit() this means that the status argument to exit() is treated in the same way as the status argument to _exit().

A language other than C may have other termination primitives than the C-language exit() function, and programs written in such a language should use its native termination primitives, but those should have as part of their function the behavior of _exit() as described. Implementations in languages other than C are outside the scope of this version of this volume of IEEE Std 1003.1-2001, however.

As required by the ISO C standard, using return from main() has the same behavior (other than with respect to language scope issues) as calling exit() with the returned value. Reaching the end of the main() function has the same behavior as calling exit(0).

A value of zero (or EXIT_SUCCESS, which is required to be zero) for the argument status conventionally indicates successful termination. This corresponds to the specification for exit() in the ISO C standard. The convention is followed by utilities such as make and various shells, which interpret a zero status from a child process as success. For this reason, applications should not call exit(0) or _exit(0) when they terminate unsuccessfully; for example, in signal-catch functions.

Historically, the implementation-defined process that inherits children whose parents have terminated without waiting on them is called init and has a process ID of 1.

The sending of a SIGHUP to the foreground process group when a controlling process terminates corresponds to somewhat different historical implementations. In System V, the kernel sends a SIGHUP on termination of (essentially) a controlling process. In 4.2 BSD, the kernel does not send SIGHUP in a case like this, but the termination of a controlling process is usually noticed by a system daemon, which arranges to send a SIGHUP to the foreground process group with the vhangup() function. However, in 4.2 BSD, due to the behavior of the shells that support job control, the controlling process is usually a shell with no other processes in its process group. Thus, a change to make _exit() behave this way in such systems should not cause problems with existing applications.

The termination of a process may cause a process group to become orphaned in either of two ways. The connection of a process group to its parent(s) outside of the group depends on both the parents and their children. Thus, a process group may be orphaned by the termination of the last connecting parent process outside of the group or by the termination of the last direct descendant of the parent process(es). In either case, if the termination of a process causes a process group to become orphaned, processes within the group are disconnected from their job control shell, which no longer has any information on the existence of the process group. Stopped processes within the group would languish forever. In order to avoid this problem, newly orphaned process groups that contain stopped processes are sent a SIGHUP signal and a SIGCONT signal to indicate that they have been disconnected from their session. The SIGHUP signal causes the process group members to terminate unless they are catching or ignoring SIGHUP. Under most circumstances, all of the members of the process group are stopped if any of them are stopped.

The action of sending a SIGHUP and a SIGCONT signal to members of a newly orphaned process group is similar to the action of 4.2 BSD, which sends SIGHUP and SIGCONT to each stopped child of an exiting process. If such children exit in response to the SIGHUP, any additional descendants receive similar treatment at that time. In this volume of IEEE Std 1003.1-2001, the signals are sent to the entire process group at the same time. Also, in
this volume of IEEE Std 1003.1-2001, but not in 4.2 BSD, stopped processes may be orphaned, but
may be members of a process group that is not orphaned; therefore, the action taken at _exit() 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>.

The ISO/IEC 9899:1999 standard adds the _Exit() function that results in immediate program
termination without triggering signals or atexit()-registered functions. In IEEE Std 1003.1-2001,
this is equivalent to the _exit() function.

FUTURE DIRECTIONS

None.

SEE ALSO

atexit(), close(), fclose(), longjmp(), posix_trace_shutdown(), posix_trace_trid_eventid_open(),
semop(), shmget(), sigaction(), wait(), waitid(), waitpid(), the Base Definitions volume of
IEEE Std 1003.1-2001, <stdlib.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.

Interactions with the SA_NOCLDWAIT flag and SIGCHLD signal are further clarified.

The values of status from exit() are better described.

Issue 6

Extensions beyond the ISO C standard are marked.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for
typed memory.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

• The _Exit() function is included.

• The DESCRIPTION is updated.

The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

References to the wait3() function are removed.
exp() — exponential function

NAME
exp, expf, expl — exponential function

SYNOPSIS
#include <math.h>

double exp(double x);
float expf(float x);
long double expl(long double x);

DESCRIPTION
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.

These functions shall compute the base-e exponential of x.

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.

RETURN VALUE
Upon successful completion, these functions shall return the exponential value of x.

If the correct value would cause overflow, a range error shall occur and exp(), expf(), and expl() shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

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.

If x is NaN, a NaN shall be returned.

If x is ±0, 1 shall be returned.

If x is −Inf, +0 shall be returned.

If x is +Inf, x shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

ERRORS
These functions shall fail if:

Range Error The result overflows.

If the integer expression (math_errno & MATH_ERRNO) is non-zero, then errno shall be set to [ERANGE]. If the integer expression (math_errno & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

Range Error The result underflows.

If the integer expression (math_errno & MATH_ERRNO) is non-zero, then errno shall be set to [ERANGE]. If the integer expression (math_errno & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.
EXAMPLES
None.

APPLICATION USAGE
Note that for IEEE Std 754-1985 double, $709.8 < x$ implies $\exp(x)$ has overflowed. The value $x < -708.4$ implies $\exp(x)$ has underflowed.

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

RATIONALE
None.

FUTURE DIRECTIONS
None.

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

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The expf() and expl() functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.
NAME
exp2, exp2f, exp2l — exponential base 2 functions

SYNOPSIS
#include <math.h>

double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);

DESCRIPTION
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.

These functions shall compute the base-2 exponential of \(x\).

An application wishing to check for error situations should set \(errno\) to zero and call \text{fexceptexcept}(FE_ALL_EXCEPT) before calling these functions. On return, if \(errno\) is non-zero or \text{fetestexcept}(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE
Upon successful completion, these functions shall return \(2^x\).

If the correct value would cause overflow, a range error shall occur and \(\text{exp2()}\), \(\text{exp2f()}\), and \(\text{exp2l()}\) shall return the value of the macro \text{HUGE_VAL}, \text{HUGE_VALF}, and \text{HUGE_VALL}, respectively.

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.

If \(x\) is NaN, a NaN shall be returned.

If \(x\) is \(\pm 0\), 1 shall be returned.

If \(x\) is \(-\text{Inf}\), 0 shall be returned.

If \(x\) is \(+\text{Inf}\), \(x\) shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

ERRORS
These functions shall fail if:

Range Error The result overflows.

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.

These functions may fail if:

Range Error The result underflows.

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.
EXAMPLES
None.

APPLICATION USAGE
For IEEE Std 754-1985 double, \(1024 \leq x\) implies \(\text{exp2}(x)\) has overflowed. The value \(x < -1022\) implies \(\text{exp}(x)\) has underflowed.

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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
\(\text{exp}()\), \(\text{feclearexcept}()\), \(\text{fetestexcept}()\), \(\text{isnan}()\), \(\text{log}()\), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, \(<\text{math.h}>>

CHANGE HISTORY
expm1()  

NAME
expm1, expm1f, expm1l — compute exponential functions

SYNOPSIS
#include <math.h>
double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);

DESCRIPTION
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

These functions shall compute \( e^x - 1.0 \).

An application wishing to check for error situations should set \( \text{errno} \) to zero and call
\text{fclearexcept}(FE_ALL_EXCEPT) before calling these functions. On return, if \( \text{errno} \) is non-zero or
\text{fetestexcept}(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE
Upon successful completion, these functions return \( e^x - 1.0 \).

If the correct value would cause overflow, a range error shall occur and \( \text{expm1()} \), \( \text{expm1f()} \), and
\( \text{expm1l()} \) shall return the value of the macro \( \text{HUGE_VAL} \), \( \text{HUGE_VALF} \), and \( \text{HUGE_VALL} \),
respectively.

If \( x \) is \( \text{NaN} \), a \( \text{NaN} \) shall be returned.

If \( x \) is \( \pm 0 \), \( \pm 0 \) shall be returned.

If \( x \) is \( -\infty \), \(-1\) shall be returned.

If \( x \) is \( +\infty \), \( x \) shall be returned.

If \( x \) is subnormal, a range error may occur and \( x \) should be returned.

ERRORS
These functions shall fail if:

Range Error   The result overflows.

If the integer expression (\( \text{math_errhandling} \& \text{MATH_ERRNO} \)) is non-zero,
then \( \text{errno} \) shall be set to [ERANGE]. If the integer expression
\( \text{math_errhandling} \& \text{MATH_ERREXCEPT} \) is non-zero, then the overflow
floating-point exception shall be raised.

These functions may fail if:

Range Error   The value of \( x \) is subnormal.

If the integer expression (\( \text{math_errhandling} \& \text{MATH_ERRNO} \)) is non-zero,
then \( \text{errno} \) 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.
**EXAMPLES**
None.

**APPLICATION USAGE**

The value of `expm1(x)` may be more accurate than `exp(x)−1.0` for small values of `x`.

The `expm1()` and `log1p()` functions are useful for financial calculations of `(1+x)^n−1)/x`, namely:

\[
\expm1(n \times \log1p(x))/x
\]

when `x` is very small (for example, when calculating small daily interest rates). These functions also simplify writing accurate inverse hyperbolic functions.

For IEEE Std 754-1985 **double**, `709.8 < x` implies `expm1(x)` has overflowed.

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

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`exp()`, `feclearexcept()`, `fetestexcept()`, `ilogb()`, `log1p()`, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, `<math.h>`

**CHANGE HISTORY**

First released in Issue 4, Version 2.

**Issue 5**
Moved from X/OPEN UNIX extension to BASE.

**Issue 6**
The `expm1f()` and `expm1l()` functions are added for alignment with the ISO/IEC 9899:1999 standard.

The `expm1()` function is no longer marked as an extension.

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

NAME
fabs, fabsf, fabsl — absolute value function

SYNOPSIS
#include <math.h>

double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);

DESCRIPTION
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.

These functions shall compute the absolute value of their argument \( x, |x| \).

RETURN VALUE
Upon successful completion, these functions shall return the absolute value of \( x \).

If \( x \) is NaN, a NaN shall be returned.

If \( x \) is ±0, +0 shall be returned.

If \( x \) is ±Inf, +Inf shall be returned.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
isinf(), the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The \( \text{fabsf}() \) and \( \text{fabsl}() \) functions are added for alignment with the ISO/IEC 9899:1999 standard.

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

NAME
fattach — attach a STREAMS-based file descriptor to a file in the file system name space (STREAMS)

SYNOPSIS
XSR
#include <stropts.h>

int fattach(int fildes, const char *path);

DESCRIPTION
The fattach() function shall attach a STREAMS-based file descriptor to a file, effectively associating a pathname with fildes. The application shall ensure that the fildes argument is a valid open file descriptor associated with a STREAMS file. The path argument points to a pathname of an existing file. The application shall have the appropriate privileges or be the owner of the file named by path and have write permission. A successful call to fattach() shall cause all pathnames that name the file named by path to name the STREAMS file associated with fildes, until the STREAMS file is detached from the file. A STREAMS file can be attached to more than one file and can have several pathnames associated with it.

The attributes of the named STREAMS file shall be initialized as follows: the permissions, user ID, group ID, and times are set to those of the file named by path, the number of links is set to 1, and the size and device identifier are set to those of the STREAMS file associated with fildes. If any attributes of the named STREAMS file are subsequently changed (for example, by chmod()), neither the attributes of the underlying file nor the attributes of the STREAMS file to which fildes refers shall be affected.

File descriptors referring to the underlying file, opened prior to an fattach() call, shall continue to refer to the underlying file.

RETURN VALUE
Upon successful completion, fattach() shall return 0. Otherwise, −1 shall be returned and errno set to indicate the error.

ERRORS
The fattach() function shall fail if:

[EACCES] Search permission is denied for a component of the path prefix, or the process is the owner of path but does not have write permissions on the file named by path.

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

[EBUSY] The file named by path is currently a mount point or has a STREAMS file attached to it.

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

[ENAMETOOLONG] The size of path exceeds [PATH_MAX] or a component of path is longer than [NAME_MAX].

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

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

[EPERM] The effective user ID of the process is not the owner of the file named by path and the process does not have appropriate privilege.
The `fattach()` function may fail if:

- **[EINVAL]** The `fildes` argument does not refer to a STREAMS file.
- **[ELOOP]** More than `{SYMLOOP_MAX}` symbolic links were encountered during resolution of the `path` argument.
- **[ENAMETOOLONG]** Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `{PATH_MAX}`.
- **[EXDEV]** A link to a file on another file system was attempted.

### EXAMPLES

**Attaching a File Descriptor to a File**

In the following example, `fd` refers to an open STREAMS file. The call to `fattach()` associates this STREAM with the file `/tmp/named-STREAM`, such that any future calls to open `/tmp/named-STREAM`, prior to breaking the attachment via a call to `fdetach()`, will instead create a new file handle referring to the STREAMS file associated with `fd`.

```c
#include <stropts.h>
... 
int fd;
char *filename = "/tmp/named-STREAM";
int ret;
ret = fattach(fd, filename);
```

**APPLICATION USAGE**

The `fattach()` function behaves similarly to the traditional `mount()` function in the way a file is temporarily replaced by the root directory of the mounted file system. In the case of `fattach()`, the replaced file need not be a directory and the replacing file is a STREAMS file.

**RATIONALE**

The file attributes of a file which has been the subject of an `fattach()` call are specifically set because of an artefact of the original implementation. The internal mechanism was the same as for the `mount()` function. Since `mount()` is typically only applied to directories, the effects when applied to a regular file are a little surprising, especially as regards the link count which rigidly remains one, even if there were several links originally and despite the fact that all original links refer to the STREAM as long as the `fattach()` remains in effect.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`fdetach()`, `isastream()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stropts.h>`

**CHANGE HISTORY**

First released in Issue 4, Version 2.

**Issue 5**

Moved from X/OPEN UNIX extension to BASE.

The [EXDEV] error is added to the list of optional errors in the ERRORS section.
This function is marked as part of the XSI STREAMS Option Group.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.
NAME
fchdir — change working directory

SYNOPSIS
XSI
#include <unistd.h>

int fchdir(int fildes);

DESCRIPTION
The fchdir() function shall be equivalent to chdir() except that the directory that is to be the new
current working directory is specified by the file descriptor fildes.
A conforming application can obtain a file descriptor for a file of type directory using open(),
provided that the file status flags and access modes do not contain O_WRONLY or O_RDWR.

RETURN VALUE
Upon successful completion, fchdir() shall return 0. Otherwise, it shall return −1 and set errno to
indicate the error. On failure the current working directory shall remain unchanged.

ERRORS
The fchdir() function shall fail if:
[EACCES] Search permission is denied for the directory referenced by fildes.
[EBADF] The fildes argument is not an open file descriptor.
[ENOTDIR] The open file descriptor fildes does not refer to a directory.
The fchdir() may fail if:
[EINTR] A signal was caught during the execution of fchdir().
[EIO] An I/O error occurred while reading from or writing to the file system.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
chdir(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
fchmod( )

NAME
fchmod — change mode of a file

SYNOPSIS
#include <sys/stat.h>
int fchmod(int fildes, mode_t mode);

DESCRIPTION
The fchmod() function shall be equivalent to chmod() except that the file whose permissions are
changed is specified by the file descriptor fildes.

SHM If fildes references a shared memory object, the fchmod() function need only affect the S_IRUSR,
S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits.

TYM If fildes references a typed memory object, the behavior of fchmod() is unspecified.

XSR If fildes refers to a STREAM (which is fattach()-ed into the file system name space) the call
returns successfully, doing nothing.

RETURN VALUE
Upon successful completion, fchmod() shall return 0. Otherwise, it shall return -1 and set errno to
indicate the error.

ERRORS
The fchmod() function shall fail if:

[EBADF] The fildes argument is not an open file descriptor.

[EPERM] The effective user ID does not match the owner of the file and the process
does not have appropriate privilege.

[EROFS] The file referred to by fildes resides on a read-only file system.

The fchmod() function may fail if:

[XSI] [EINTR] The fchmod() function was interrupted by a signal.

[XSI] [EINVAL] The value of the mode argument is invalid.

[XSI] [EINVAL] The fildes argument refers to a pipe and the implementation disallows
execution of fchmod() on a pipe.

EXAMPLES
Changing the Current Permissions for a File

The following example shows how to change the permissions for a file named /home/cnd/mod1
so that the owner and group have read/write/execute permissions, but the world only has
read/write permissions.

#include <sys/stat.h>
#include <fcntl.h>
mode_t mode;
int fildes;
... fildes = open("/home/cnd/mod1", O_RDWR);
fchmod(fildes, S_IRWXU | S_IRWXG | S_IROTH | S_IWOTH);
fchmod()

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
chmod(), chown(), creat(), fcntl(), fstatvfs(), mknod(), open(), read(), stat(), write(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/stat.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE and aligned with fchmod() in the POSIX Realtime Extension. Specifically, the second paragraph of the DESCRIPTION is added and a second instance of [EINVAL] is defined in the list of optional errors.

Issue 6
The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by stating that fchmod() behavior is unspecified for typed memory objects.
NAME
fchown — change owner and group of a file

SYNOPSIS
#include <unistd.h>
int fchown(int fildes, uid_t owner, gid_t group);

DESCRIPTION
The fchown() function shall be equivalent to chown() except that the file whose owner and group are changed is specified by the file descriptor fildes.

RETURN VALUE
Upon successful completion, fchown() shall return 0. Otherwise, it shall return −1 and set errno to indicate the error.

ERRORS
The fchown() function shall fail if:

- [EBADF] The fildes argument is not an open file descriptor.
- [EPERM] The effective user ID does not match the owner of the file or the process does not have appropriate privilege and _POSIX_CHOWN_RESTRICTED indicates that such privilege is required.
- [EROFS] The file referred to by fildes resides on a read-only file system.

The fchown() function may fail if:

- [EINVAL] The owner or group ID is not a value supported by the implementation. The fildes argument refers to a pipe or socket or an fattach()-ed STREAM and the implementation disallows execution of fchown() on a pipe.
- [EIO] A physical I/O error has occurred.
- [EINTR] The fchown() function was interrupted by a signal which was caught.

EXAMPLES

Changing the Current Owner of a File
The following example shows how to change the owner of a file named /home/cnd/mod1 to "jones" and the group to "cnd".

The numeric value for the user ID is obtained by extracting the user ID from the user database entry associated with "jones". Similarly, the numeric value for the group ID is obtained by extracting the group ID from the group database entry associated with "cnd". This example assumes the calling program has appropriate privileges.

#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <pwd.h>
#include <grp.h>

struct passwd *pwd;
struct group *grp;
int fildes;
...
fildes = open("/home/cnd/mod1", 0_RDWR);
pwd = getpwnam("jones");
fchown()  

grp = getgrnam("cnd");  
fchown(fildes, pwd->pw_uid, grp->gr_gid);

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
chown(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
The following changes were made to align with the IEEE P1003.1a draft standard:
• Clarification is added that a call to fchown() may not be allowed on a pipe.
The fchown() function is now defined as mandatory.
NAME
fclose — close a stream

SYNOPSIS
#include <stdio.h>

int fclose(FILE *stream);

DESCRIPTION
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
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.

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.

After the call to fclose(), any use of stream results in undefined behavior.

RETURN VALUE
Upon successful completion, fclose() shall return 0; otherwise, it shall return EOF and set errno to
indicate the error.

ERRORS
The fclose() function shall fail if:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying stream and the
process would be delayed in the write operation.

[EBADF] The file descriptor underlying stream is not valid.

[EFBIG] An attempt was made to write a file that exceeds the maximum file size.

(EIFBIG] An attempt was made to write a file that exceeds the process’ file size limit.

[ENOSPC] There was no free space remaining on the device containing the file.

[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.

The fclose() function may fail if:

[ENXIO] A request was made of a nonexistent device, or the request was outside the
capabilities of the device.
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
close(), fopen(), getrlimit(), ulimit(), the Base Definitions volume of IEEE Std 1003.1-2001,
<stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
Large File Summit extensions are added.

Issue 6
Extensions beyond the ISO C standard are marked.

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

• The [EFBIG] error is added as part of the large file support extensions.
• The [ENXIO] optional error condition is added.

The DESCRIPTION is updated to note that the stream and any buffer are disassociated whether
or not the call succeeds. This is for alignment with the ISO/IEC 9899:1999 standard.
NAME
fcntl — file control

SYNOPSIS
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fildes, int cmd, ...);

DESCRIPTION
The fcntl() function shall perform the operations described below on open files. The fildes argument is a file descriptor.

The available values for cmd are defined in <fcntl.h> and are as follows:

- **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, *arg*, 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 exec functions.

- **F_GETFD** Get the file descriptor flags defined in <fcntl.h> that are associated with the file descriptor *fildes*. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file.

- **F_SETFD** Set the file descriptor flags defined in <fcntl.h>, that are associated with *fildes*, to the third argument, *arg*, taken as type int. If the FD_CLOEXEC flag in the third argument is 0, the file shall remain open across the exec functions; otherwise, the file shall be closed upon successful execution of one of the exec functions.

- **F_GETFL** Get the file status flags and file access modes, defined in <fcntl.h>, for the file description associated with *fildes*. 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.

- **F_SETFL** Set the file status flags, defined in <fcntl.h>, for the file description associated with *fildes* from the corresponding bits in the third argument, *arg*, 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 *arg* shall be ignored. If any bits in *arg* other than those mentioned here are changed by the application, the result is unspecified.

- **F_GETOWN** If *fildes* 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 *fildes* does not refer to a socket, the results are unspecified.

- **F_SETOWN** If *fildes* 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, *arg*, taken as type int. Positive values indicate a process ID; negative values, other than −1, indicate a process group ID. If *fildes* does not refer to a socket, the results are unspecified.
The following values for \textit{cmd} are available for advisory record locking. Record locking shall be supported for regular files, and may be supported for other files.

\begin{itemize}
\item **F_GETLK** Get the first lock which blocks the lock description pointed to by the third argument, \textit{arg}, taken as a pointer to type \texttt{struct flock}, defined in \texttt{<fcntl.h>}. The information retrieved shall overwrite the information passed to \texttt{fcntl()} in the structure \texttt{flock}. If no lock is found that would prevent this lock from being created, then the structure shall be left unchanged except for the lock type which shall be set to F_UNLCK.
\item **F_SETLK** Set or clear a file segment lock according to the lock description pointed to by the third argument, \textit{arg}, taken as a pointer to type \texttt{struct flock}, defined in \texttt{<fcntl.h>}. F_SETLK can establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks (F_WRLCK), as well as to remove either type of lock (F_UNLCK). F_RDLCK, F_WRLCK, and F_UNLCK are defined in \texttt{<fcntl.h>}. If a shared or exclusive lock cannot be set, \texttt{fcntl()} shall return immediately with a return value of \texttt{-1}.
\item **F_SETLKW** This command shall be equivalent to F_SETLK except that if a shared or exclusive lock is blocked by other locks, the thread shall wait until the request can be satisfied. If a signal that is to be caught is received while \texttt{fcntl()} is waiting for a region, \texttt{fcntl()} shall be interrupted. Upon return from the signal handler, \texttt{fcntl()} shall return \texttt{-1} with \texttt{errno} set to [EINTR], and the lock operation shall not be done.
\end{itemize}

Additional implementation-defined values for \textit{cmd} may be defined in \texttt{<fcntl.h>}. Their names shall start with \texttt{F}_.

When a shared lock is set on a segment of a file, other processes shall be able to set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock shall fail if the file descriptor was not opened with read access.

An exclusive lock shall prevent any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock shall fail if the file descriptor was not opened with write access.

The structure \texttt{flock} describes the type (\texttt{l_type}), starting offset (\texttt{l_whence}), relative offset (\texttt{l_start}), size (\texttt{l_len}), and process ID (\texttt{l_pid}) of the segment of the file to be affected.

The value of \texttt{l_whence} is SEEK_SET, SEEK_CUR, or SEEK_END, to indicate that the relative offset \texttt{l_start} bytes shall be measured from the start of the file, current position, or end of the file, respectively. The value of \texttt{l_len} is the number of consecutive bytes to be locked. The value of \texttt{l_len} may be negative (where the definition of \texttt{off_t} permits negative values of \texttt{l_len}). The \texttt{l_pid} field is only used with F_GETLK to return the process ID of the process holding a blocking lock. After a successful F_GETLK request, when a blocking lock is found, the values returned in the \texttt{flock} structure shall be as follows:

\begin{itemize}
\item \texttt{l_type} Type of blocking lock found.
\item \texttt{l_whence} SEEK_SET.
\item \texttt{l_start} Start of the blocking lock.
\item \texttt{l_len} Length of the blocking lock.
\item \texttt{l_pid} Process ID of the process that holds the blocking lock.
\end{itemize}
If the command is F_SETLK 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_UNLOCK) 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_UNLOCK) request shall attempt to unlock only the requested segment.

When the file descriptor `fildes` 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.

If `fildes` refers to a typed memory object, the result of the `fcntl()` function is unspecified.

**RETURN VALUE**

Upon successful completion, the value returned shall depend on `cmd` as follows:

- **F_DUPFD** A new file descriptor.
- **F_GETFD** Value of flags defined in `<fcntl.h>`. The return value shall not be negative.
- **F_SETFD** Value other than −1.
- **F_GETFL** Value of file status flags and access modes. The return value is not negative.
- **F_SETFL** Value other than −1.
- **F_GETLK** Value other than −1.
- **F_SETLK** Value other than −1.
- **F_SETLKW** Value other than −1.
- **F_GETOWN** Value of the socket owner process or process group; this will not be −1.
The `fcntl()` function shall fail if:

- **[EACCES]** The `cmd` argument is `F_SETLK`; the type of lock (`l_type`) is a shared (F_RDLCK) or exclusive (F_WRLCK) lock and the segment of a file to be locked is already exclusive-locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

- **[EBADF]** The `fildes` argument is not a valid open file descriptor, or the argument `cmd` is `F_SETLK` or `F_SETLKW`, the type of lock, `l_type`, is a shared lock (F_RDLCK), and `fildes` is not a valid file descriptor open for reading, or the type of lock, `l_type`, is an exclusive lock (F_WRLCK), and `fildes` is not a valid file descriptor open for writing.

- **[EINTR]** The `cmd` argument is `F_SETLKW` and the function was interrupted by a signal.

- **[EINVAL]** The `cmd` argument is invalid, or the `cmd` argument is `F_DUPFD` and `arg` is negative or greater than or equal to `{OPEN_MAX}`, or the `cmd` argument is `F_GETLK`, `F_SETLK`, or `F_SETLKW` and the data pointed to by `arg` is not valid, or `fildes` refers to a file that does not support locking.

- **[EMFILE]** The `cmd` argument is `F_DUPFD` and `{OPEN_MAX}` file descriptors are currently open in the calling process, or no file descriptors greater than or equal to `arg` are available.

- **[ENOLCK]** The argument `cmd` is `F_SETLK` or `F_SETLKW` and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

- **[EOVERFLOW]** One of the values to be returned cannot be represented correctly.

- **[EOVERFLOW]** The `cmd` argument is `F_GETLK`, `F_SETLK`, or `F_SETLKW` and the smallest or, if `l_len` is non-zero, the largest offset of any byte in the requested segment cannot be represented correctly in an object of type `off_t`.

The `fcntl()` function may fail if:

- **[EDEADLK]** The `cmd` argument is `F_SETLKW`, the lock is blocked by a lock from another process, and putting the calling process to sleep to wait for that lock to become free would cause a deadlock.

The ellipsis in the SYNOPSIS is the syntax specified by the ISO C standard for a variable number of arguments. It is used because System V uses pointers for the implementation of file locking functions.

The `arg` values to `F_GETFD`, `F_SETFD`, `F_GETFL`, and `F_SETFL` all represent flag values to allow for future growth. Applications using these functions should do a read-modify-write operation.
System Interfaces

fcntl( )

10872 on them, rather than assuming that only the values defined by this volume of
10873 IEEE Std 1003.1-2001 are valid. It is a common error to forget this, particularly in the case of
10874 F_SETFD.

10875 This volume of IEEE Std 1003.1-2001 permits concurrent read and write access to file data using
10876 the fcntl() function; this is a change from the 1984 /usr/group standard and early proposals.
10877 Without concurrency controls, this feature may not be fully utilized without occasional loss of
10878 data.

10879 Data losses occur in several ways. One case occurs when several processes try to update the
10880 same record, without sequencing controls; several updates may occur in parallel and the last
10881 writer “wins”. Another case is a bit-tree or other internal list-based database that is undergoing
10882 reorganization. Without exclusive use to the tree segment by the updating process, other reading
10883 processes chance getting lost in the database when the index blocks are split, condensed,
10884 inserted, or deleted. While fcntl() is useful for many applications, it is not intended to be overly
10885 general and does not handle the bit-tree example well.

10886 This facility is only required for regular files because it is not appropriate for many devices such
10887 as terminals and network connections.

10888 Since fcntl() works with “any file descriptor associated with that file, however it is obtained”,
10889 the file descriptor may have been inherited through a fork() or exec operation and thus may
10890 affect a file that another process also has open.

10891 The use of the open file description to identify what to lock requires extra calls and presents
10892 problems if several processes are sharing an open file description, but there are too many
10893 implementations of the existing mechanism for this volume of IEEE Std 1003.1-2001 to use
10894 different specifications.

10895 Another consequence of this model is that closing any file descriptor for a given file (whether or
10896 not it is the same open file description that created the lock) causes the locks on that file to be
10897 relinquished for that process. Equivalently, any close for any file/process pair relinquishes the
10898 locks owned on that file for that process. But note that while an open file description may be
10899 shared through fork(), locks are not inherited through fork(). Yet locks may be inherited through
10900 one of the exec functions.

10901 The identification of a machine in a network environment is outside the scope of this volume of
10902 IEEE Std 1003.1-2001. Thus, an l_sysid member, such as found in System V, is not included in the
10903 locking structure.

10904 Changing of lock types can result in a previously locked region being split into smaller regions.

10905 Mandatory locking was a major feature of the 1984 /usr/group standard.

10906 For advisory file record locking to be effective, all processes that have access to a file must
10907 cooperate and use the advisory mechanism before doing I/O on the file. Enforcement-mode
10908 record locking is important when it cannot be assumed that all processes are cooperating. For
10909 example, if one user uses an editor to update a file at the same time that a second user executes
10910 another process that updates the same file and if only one of the two processes is using advisory
10911 locking, the processes are not cooperating. Enforcement-mode record locking would protect
10912 against accidental collisions.

10913 Secondly, advisory record locking requires a process using locking to bracket each I/O operation
10914 with lock (or test) and unlock operations. With enforcement-mode file and record locking, a
10915 process can lock the file once and unlock when all I/O operations have been completed.
10916 Enforcement-mode record locking provides a base that can be enhanced; for example, with
10917 sharable locks. That is, the mechanism could be enhanced to allow a process to lock a file so
10918 other processes could read it, but none of them could write it.
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, \texttt{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 \texttt{[EDEADLK]} error was made optional.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

\texttt{alarm()}, \texttt{close()}, \texttt{exec()}, \texttt{open()}, \texttt{sigaction()}, the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<fcntl.h>}, \texttt{<signal.h>}, \texttt{<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 \texttt{<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 \texttt{<sys/types.h>} has been removed. Although \texttt{<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 \texttt{l_len} is negative are now mandated, and the description of unlock (F_UNLOCK) when \texttt{l_len} is non-negative is mandated.
- In the ERRORS section, the \texttt{[EINVAL]} error condition has the case mandated when the \texttt{cmd} is invalid, and two \texttt{[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 \texttt{fcntl()} results are unspecified for typed memory objects.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
fcvt — convert a floating-point number to a string (LEGACY)

SYNOPSIS
XSI
#include <stdlib.h>

char *fcvt(double value, int ndigit, int *restrict decpt,
            int *restrict sign);

DESCRIPTION
Refer to ecvt().
NAME
fdatasync — synchronize the data of a file (REALTIME)

SYNOPSIS
#include <unistd.h>
int fdatasync(int fildes);

DESCRIPTION
The fdatasync() 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.

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.

RETURN VALUE
If successful, the fdatasync() function shall return the value 0; otherwise, the function shall return the value −1 and set errno to indicate the error. If the fdatasync() function fails, outstanding I/O operations are not guaranteed to have been completed.

ERRORS
The fdatasync() function shall fail if:

[EBADF] The fildes argument is not a valid file descriptor open for writing.

[EINVAL] This implementation does not support synchronized I/O for this file.

In the event that any of the queued I/O operations fail, fdatasync() shall return the error conditions defined for read() and write().

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
aio_fsync(), fcntl(), fsync(), open(), read(), write(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

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

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

The fdatasync() function is marked as part of the Synchronized Input and Output option.
fdetach() — detach a name from a STREAMS-based file descriptor (STREAMS)

SYNOPSIS

```c
#include <stropts.h>

int fdetach(const char *path);
```

DESCRIPTION

The `fdetach()` function shall detach a STREAMS-based file from the file to which it was attached by a previous call to `fattach()`. The `path` argument points to the pathname of the attached STREAMS file. The process shall have appropriate privileges or be the owner of the file. A successful call to `fdetach()` shall cause all pathnames that named the attached STREAMS file to again name the file to which the STREAMS file was attached. All subsequent operations on `path` shall operate on the underlying file and not on the STREAMS file.

All open file descriptions established while the STREAMS file was attached to the file referenced by `path` shall still refer to the STREAMS file after the `fdetach()` has taken effect.

If there are no open file descriptors or other references to the STREAMS file, then a successful call to `fdetach()` shall be equivalent to performing the last `close()` on the attached file.

RETURN VALUE

Upon successful completion, `fdetach()` shall return 0; otherwise, it shall return −1 and set `errno` to indicate the error.

ERRORS

The `fdetach()` function shall fail if:

- [EACCES] Search permission is denied on a component of the path prefix.
- [EINVAL] The `path` argument names a file that is not currently attached.
- [ELOOP] A loop exists in symbolic links encountered during resolution of the `path` argument.
- [ENAMETOOLONG] The size of a pathname exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
- [ENOENT] A component of `path` does not name an existing file or `path` is an empty string.
- [ENOTDIR] A component of the path prefix is not a directory.
- [EPERM] The effective user ID is not the owner of `path` and the process does not have appropriate privileges.

The `fdetach()` function may fail if:

- [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the `path` argument.
- [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}. 

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
EXAMPLES

Detaching a File

The following example detaches the STREAMS-based file /tmp/named-STREAM from the file to which it was attached by a previous, successful call to fattach(). Subsequent calls to open this file refer to the underlying file, not to the STREAMS file.

```c
#include <stropts.h>
...
char *filename = "/tmp/named-STREAM";
int ret;
ret = fdetach(filename);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fattach(), the Base Definitions volume of IEEE Std 1003.1-2001, <stropts.h>

CHANGE HISTORY

Issue 5
First released in Issue 4, Version 2.

Issue 6
Moved from X/OPEN UNIX extension to BASE.

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

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.
NAME
fdim, fdimf, fdiml — compute positive difference between two floating-point numbers

SYNOPSIS
#include <math.h>

double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);

DESCRIPTION
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.

These functions shall determine the positive difference between their arguments. If \( x \) is greater than \( y \), \( x - y \) is returned. If \( x \) is less than or equal to \( y \), \(+0i\) is returned.

An application wishing to check for error situations should set \texttt{errno} to zero and call \texttt{f curexcept} (FE_ALL_EXCEPT) before calling these functions. On return, if \texttt{errno} is non-zero or \texttt{f testexcept} (FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE
Upon successful completion, these functions shall return the positive difference value. If \( x - y \) is positive and overflows, a range error shall occur and \texttt{fdim()}, \texttt{fdimf()}, and \texttt{fdiml()} shall return the value of the macro \texttt{HUGE_VAL}, \texttt{HUGE_VALF}, and \texttt{HUGE_VALL}, respectively.

If \( x - y \) is positive and underflows, a range error may occur, and either \( (x - y) \) (if representable), or \( 0.0 \) (if supported), or an implementation-defined value shall be returned.

If \( x \) or \( y \) is NaN, a NaN shall be returned.

ERRORS
The \texttt{fdim()} function shall fail if:

Range Error The result overflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then \texttt{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.

The \texttt{fdim()} function may fail if:

Range Error The result underflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then \texttt{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.
EXAMPLES

None.

APPLICATION USAGE

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.

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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>

CHANGE HISTORY

NAME
fdopen — associate a stream with a file descriptor

SYNOPSIS

```c
#include <stdio.h>

FILE *fdopen(int fildes, const char *mode);
```

DESCRIPTION

The `fdopen()` function shall associate a stream with a file descriptor.

The `mode` argument is a character string having one of the following values:

- `r` or `rb`  
  Open a file for reading.

- `w` or `wb`  
  Open a file for writing.

- `a` or `ab`  
  Open a file for writing at end-of-file.

- `r+` or `rb+` or `r+b`  
  Open a file for update (reading and writing).

- `w+` or `wb+` or `w+b`  
  Open a file for update (reading and writing).

- `a+` or `ab+` or `a+b`  
  Open a file for update (reading and writing) at end-of-file.

The meaning of these flags is exactly as specified in `fopen()`, except that modes beginning with `w` shall not cause truncation of the file.

Additional values for the `mode` argument may be supported by an implementation.

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.

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.

If `fildes` refers to a shared memory object, the result of the `fdopen()` function is unspecified.

If `fildes` refers to a typed memory object, the result of the `fdopen()` function is unspecified.

The `fdopen()` function shall preserve the offset maximum previously set for the open file description corresponding to `fildes`.

RETURN VALUE

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.

ERRORS

The `fdopen()` function may fail if:

- `[EBADF]`  
  The `fildes` argument is not a valid file descriptor.

- `[EINVAL]`  
  The `mode` argument is not a valid mode.

- `[EMFILE]`  
  (FOPEN_MAX) streams are currently open in the calling process.

- `[EMFILE]`  
  (STREAM_MAX) streams are currently open in the calling process.

- `[ENOMEM]`  
  Insufficient space to allocate a buffer.
EXCEPTIONS
None.

APPLICATION USAGE
File descriptors are obtained from calls like open(), dup(), creat(), or pipe(), which open files but do not return streams.

RATIONALE
The file descriptor may have been obtained from open(), creat(), pipe(), dup(), or fcntl(); inherited through fork() or exec; or perhaps obtained by implementation-defined means, such as the 4.3 BSD socket() call.

The meanings of the mode arguments of fdopen() and fopen() differ. With fdopen(), open for write (w or w+) does not truncate, and append (a or a+) cannot create for writing. The mode argument formats that include a b are allowed for consistency with the ISO C standard function fopen().

The b has no effect on the resulting stream. Although not explicitly required by this volume of IEEE Std 1003.1-2001, a good implementation of append (a) mode would cause the O_APPEND flag to be set.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.5.1 (on page 35), fclose(), fopen(), open(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.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.

Large File Summit extensions are added.

Issue 6
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

• In the DESCRIPTION, the use and setting of the mode argument are changed to include binary streams.

• In the DESCRIPTION, text is added for large file support to indicate setting of the offset maximum in the open file description.

• All errors identified in the ERRORS section are added.

• In the DESCRIPTION, text is added that the fdopen() function may cause st_atime to be updated.

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

• Clarification is added that it is the responsibility of the application to ensure that the mode is compatible with the open file descriptor.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that fdopen() results are unspecified for typed memory objects.
NAME  
feclearexcept — clear floating-point exception

SYNOPSIS
#include <fenv.h>
int feclearexcept(int excepts);

DESCRIPTION

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 feclearexcept() function shall attempt to clear the supported floating-point exceptions represented by excepts.

RETURN VALUE

If the argument is zero or if all the specified exceptions were successfully cleared, feclearexcept() shall return zero. Otherwise, it shall return a non-zero value.

ERRORS

No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fegetexceptflag(), feraiseexcept(), fsetexceptflag(), fetestexcept(), the Base Definitions volume of IEEE Std 1003.1-2001, <fenv.h>

CHANGE HISTORY

NAME
fegetenv, fesetenv — get and set current floating-point environment

SYNOPSIS
#include <fenv.h>

int fegetenv(fenv_t *envp);
int fesetenv(const fenv_t *envp);

DESCRIPTION
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

The fegetenv() function shall attempt to store the current floating-point environment in the object
pointed to by envp.

The fesetenv() function shall attempt to establish the floating-point environment represented by
the object pointed to by envp. The argument envp shall point to an object set by a call to
fegetenv() or feholdexcept(), or equal a floating-point environment macro. The fesetenv() function
does not raise floating-point exceptions, but only installs the state of the floating-point status
flags represented through its argument.

RETURN VALUE
If the representation was successfully stored, fegetenv() shall return zero. Otherwise, it shall
return a non-zero value. If the environment was successfully established, fesetenv() shall return
zero. Otherwise, it shall return a non-zero value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
feholdexcept(), feupdateenv(), the Base Definitions volume of IEEE Std 1003.1-2001, <fenv.h>

CHANGE HISTORY
fegetexceptflag() — get and set floating-point status flags

#include <fenv.h>

int fegetexceptflag(fexcept_t *flagp, int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts);

DESCRIPTION

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

The fegetexceptflag() function shall attempt to store an implementation-defined representation of
the states of the floating-point status flags indicated by the argument excepts in the object
pointed to by the argument flagp.

The fesetexceptflag() function shall attempt to set the floating-point status flags indicated by the
argument excepts to the states stored in the object pointed to by flagp. The value pointed to by
flagp shall have been set by a previous call to fegetexceptflag() whose second argument
represented at least those floating-point exceptions represented by the argument excepts. This
function does not raise floating-point exceptions, but only sets the state of the flags.

RETURN VALUE

If the representation was successfully stored, fegetexceptflag() shall return zero. Otherwise, it
shall return a non-zero value. If the excepts argument is zero or if all the specified exceptions
were successfully set, fesetexceptflag() shall return zero. Otherwise, it shall return a non-zero
value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept(), feraiseexcept(), fetestexcept(), the Base Definitions volume of IEEE Std 1003.1-2001,
<fenv.h>

CHANGE HISTORY

NAME
fegetround, fesetround — get and set current rounding direction

SYNOPSIS
#include <fenv.h>

int fegetround(void);
int fesetround(int round);

DESCRIPTION
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
The fegetround() function shall get the current rounding direction.
The fesetround() function shall establish the rounding direction represented by its argument
round. If the argument is not equal to the value of a rounding direction macro, the rounding
direction is not changed.

RETURN VALUE
The fegetround() function shall return the value of the rounding direction macro representing the
current rounding direction or a negative value if there is no such rounding direction macro or
the current rounding direction is not determinable.
The fesetround() function shall return a zero value if and only if the requested rounding direction
was established.

ERRORS
No errors are defined.

EXAMPLES
The following example saves, sets, and restores the rounding direction, reporting an error and
aborting if setting the rounding direction fails:

#include <fenv.h>
#include <cassert.h>
void f(int round_dir)
{
    #pragma STDC FENV_ACCESS ON
    int save_round;
    int setround_ok;
    save_round = fegetround();
    setround_ok = fesetround(round_dir);
    assert(setround_ok == 0);
    /* ... */
    fesetround(save_round);
    /* ... */
}

APPLICATION USAGE
None.

RATIONALE
None.
FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <fenv.h>

CHANGE HISTORY
NAME
feholdexcept — save current floating-point environment

SYNOPSIS
#include <fenv.h>

int feholdexcept(fenv_t *envp);

DESCRIPTION
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

The feholdexcept() function shall save the current floating-point environment in the object
pointed to by envp, clear the floating-point status flags, and then install a non-stop (continue on
floating-point exceptions) mode, if available, for all floating-point exceptions.

RETURN VALUE
The feholdexcept() function shall return zero if and only if non-stop floating-point exception
handling was successfully installed.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
The feholdexcept() function should be effective on typical IEC 60559: 1989 standard
implementations which have the default non-stop mode and at least one other mode for trap
handling or aborting. If the implementation provides only the non-stop mode, then installing the
non-stop mode is trivial.

FUTURE DIRECTIONS
None.

SEE ALSO
fegetenv(), fesetenv(), feupdateenv(), the Base Definitions volume of IEEE Std 1003.1-2001,
<fenv.h>

CHANGE HISTORY
NAME
feof — test end-of-file indicator on a stream

SYNOPSIS
#include <stdio.h>
int feof(FILE *stream);

DESCRIPTION
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
The feof() function shall test the end-of-file indicator for the stream pointed to by stream.

RETURN VALUE
The feof() function shall return non-zero if and only if the end-of-file indicator is set for stream.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
clearerr(), ferror(), fopen(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
feraiseexcept — raise floating-point exception

SYNOPSIS
#include <fenv.h>
int feraiseexcept(int excepts);

DESCRIPTION
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

The feraiseexcept() function shall attempt to raise the supported floating-point exceptions
represented by the argument excepts. The order in which these floating-point exceptions are
raised is unspecified. Whether the feraiseexcept() function additionally raises the inexact
floating-point exception whenever it raises the overflow or underflow floating-point exception is
implementation-defined.

RETURN VALUE
If the argument is zero or if all the specified exceptions were successfully raised, feraiseexcept()
shall return zero. Otherwise, it shall return a non-zero value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
The effect is intended to be similar to that of floating-point exceptions raised by arithmetic
operations. Hence, enabled traps for floating-point exceptions raised by this function are taken.

RATIONALE
Raising overflow or underflow is allowed to also raise inexact because on some architectures the
only practical way to raise an exception is to execute an instruction that has the exception as a
side effect. The function is not restricted to accept only valid coincident expressions for atomic
operations, so the function can be used to raise exceptions accrued over several operations.

FUTURE DIRECTIONS
None.

SEE ALSO
fclearexcept(), fetexceptflag(), fetsetexceptflag(), fetestexcept(), the Base Definitions volume of
IEEE Std 1003.1-2001, <fenv.h>

CHANGE HISTORY
NAME
ferror — test error indicator on a stream

SYNOPSIS
#include <stdio.h>
int ferror(FILE *stream);

DESCRIPTION
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
The ferror() function shall test the error indicator for the stream pointed to by stream.

RETURN VALUE
The ferror() function shall return non-zero if and only if the error indicator is set for stream.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
clearerr(), feof(), fopen(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
fesetenv — set current floating-point environment

SYNOPSIS

#include <fenv.h>

int fesetenv(const fenv_t *envp);

DESCRIPTION

Refer to fegetenv().
NAME
  fesetexceptflag — set floating-point status flags

SYNOPSIS
  #include <fenv.h>
  int fesetexceptflag(const fexcept_t *flagp, int excepts);

DESCRIPTION
  Refer to fegetexceptflag().
NAME
fesetround — set current rounding direction

SYNOPSIS
#include <fenv.h>

int fesetround(int round);

DESCRIPTION
Refer to fegetround().
NAME
fetestexcept — test floating-point exception flags

SYNOPSIS
#include <fenv.h>

int fetestexcept(int excepts);

DESCRIPTION
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 fetestexcept() function shall determine which of a specified subset of the floating-point exception flags are currently set. The excepts argument specifies the floating-point status flags to be queried.

RETURN VALUE
The fetestexcept() function shall return the value of the bitwise-inclusive OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in excepts.

ERRORS
No errors are defined.

EXAMPLES
The following example calls function f() if an invalid exception is set, and then function g() if an overflow exception is set:

#include <fenv.h>
/* ... */
{
 # pragma STDC FENV_ACCESS ON
  int set_excepts;
  feclearexcept(FE_INVALID | FE_OVERFLOW);
  // maybe raise exceptions
  set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
  if (set_excepts & FE_INVALID) f();
  if (set_excepts & FE_OVERFLOW) g();
  /* ... */
}

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
feclearexcept(), fegetexceptflag(), feraiseexcept(), the Base Definitions volume of IEEE Std 1003.1-2001, <fenv.h>
CHANGE HISTORY
NAME
feupdateenv — update floating-point environment

SYNOPSIS
#include <fenv.h>

int feupdateenv(const fenv_t *envp);

DESCRIPTION
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 feupdateenv() function shall attempt to save the currently raised floating-point exceptions in its automatic storage, attempt to install the floating-point environment represented by the object pointed to by envp, and then attempt to raise the saved floating-point exceptions. The argument envp shall point to an object set by a call to feholdexcept() or fegetenv(), or equal a floating-point environment macro.

RETURN VALUE
The feupdateenv() function shall return a zero value if and only if all the required actions were successfully carried out.

ERRORS
No errors are defined.

EXAMPLES
The following example shows sample code to hide spurious underflow floating-point exceptions:

```c
#include <fenv.h>

double f(double x)
{
  #pragma STDC FENV_ACCESS ON
  double result;
  fenv_t save_env;
  feholdexcept(&save_env);
  // compute result
  if (/* test spurious underflow */) 
    feclearexcept(FE_UNDERFLOW);
  feupdateenv(&save_env);
  return result;
}
```

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fegetenv(), feholdexcept(), the Base Definitions volume of IEEE Std 1003.1-2001, <fenv.h>
CHANGE HISTORY


NAME
fflush — flush a stream

SYNOPSIS
#include <stdio.h>

int fflush(FILE *stream);

DESCRIPTION
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

If stream points to an output stream or an update stream in which the most recent operation was
not input, fflush() shall cause any unwritten data for that stream to be written to the file, and the
st_ctime and st_mtime fields of the underlying file shall be marked for update.

If stream is a null pointer, fflush() shall perform this flushing action on all streams for which the
behavior is defined above.

RETURN VALUE
Upon successful completion, fflush() shall return 0; otherwise, it shall set the error indicator for
the stream, return EOF, and set errno to indicate the error.

ERRORS
The fflush() function shall fail if:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying stream and the
process would be delayed in the write operation.

[EBADF] The file descriptor underlying stream is not valid.

[EFBIG] An attempt was made to write a file that exceeds the maximum file size.

[EFBIG] An attempt was made to write a file that exceeds the process’ file size limit.

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.

[EINTR] The fflush() function was interrupted by a signal.

[ENOSPC] There was no free space remaining on the device containing the file.

[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.

The fflush() function may fail if:

[ENXIO] A request was made of a nonexistent device, or the request was outside the
capabilities of the device.
**EXAMPLES**

**Sending Prompts to Standard Output**

The following example uses `printf()` calls to print a series of prompts for information the user must enter from standard input. The `fflush()` calls force the output to standard output. The `fflush()` function is used because standard output is usually buffered and the prompt may not immediately be printed on the output or terminal. The `gets()` calls read strings from standard input and place the results in variables, for use later in the program.

```c
#include <stdio.h>
...
char user[100];
char oldpasswd[100];
char newpasswd[100];
...
printf("User name: ");
fflush(stdout);
gets(user);
printf("Old password: ");
fflush(stdout);
gets(oldpasswd);
printf("New password: ");
fflush(stdout);
gets(newpasswd);
...
```

**APPLICATION USAGE**

None.

**RATIONALE**

Data buffered by the system may make determining the validity of the position of the current file descriptor impractical. Thus, enforcing the repositioning of the file descriptor after `fflush()` on streams open for `read()` is not mandated by IEEE Std 1003.1-2001.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`getrlimit()`, `ulimit()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`

**CHANGE HISTORY**

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

**Issue 5**

Large File Summit extensions are added.

**Issue 6**

Extensions beyond the ISO C standard are marked.

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

- The [EFBIG] error is added as part of the large file support extensions.
- The [ENXIO] optional error condition is added.
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.
NAME
ffs — find first set bit

SYNOPSIS
XSI
#include <strings.h>

int ffs(int i);

DESCRIPTION
The ffs() function shall find the first bit set (beginning with the least significant bit) in i, and return the index of that bit. Bits are numbered starting at one (the least significant bit).

RETURN VALUE
The ffs() function shall return the index of the first bit set. If i is 0, then ffs() shall return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
NAME
fgetc — get a byte from a stream

SYNOPSIS
#include <stdio.h>
int fgetc(FILE *stream);

DESCRIPTION
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

If the end-of-file indicator for the input stream pointed to by stream is not set and a next byte is
present, the fgetc() function shall obtain the next byte as an unsigned char converted to an int,
from the input stream pointed to by stream, and advance the associated file position indicator for
the stream (if defined). Since fgetc() operates on bytes, reading a character consisting of multiple
bytes (or “a multi-byte character”) may require multiple calls to fgetc().

The fgetc() 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(),
fgetws(), fread(), fscanf(), getc(), getchar(), gets(), or scanf() using stream that returns
data not supplied by a prior call to ungetc() or ungetwc().

RETURN VALUE
Upon successful completion, fgetc() shall return the next byte from the input stream pointed to
by stream. If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the
error indicator for the stream shall be set and fgetc() shall return EOF. If a read error occurs,
the error indicator for the stream shall be set, fgetc() shall return EOF, and shall set errno to
indicate the error.

ERRORS
The fgetc() function shall fail if data needs to be read and:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying stream and the
process would be delayed in the fgetc() operation.

[EBADF] The file descriptor underlying stream is not a valid file descriptor open for
reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and no data
was transferred.

[EIO] A physical I/O error has occurred, or the process is in a background process
group attempting to read from its controlling terminal, and either the process
is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
This error may also be generated for implementation-defined reasons.

[EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the
offset maximum associated with the corresponding stream.

The fgetc() function may fail if:

[ENOMEM] Insufficient storage space is available.

[ENXIO] A request was made of a nonexistent device, or the request was outside the
capabilities of the device.
EXAMPLES

None.

APPLICATION USAGE

If the integer value returned by \texttt{fgetc()} is stored into a variable of type \texttt{char} and then compared against the integer constant EOF, the comparison may never succeed, because sign-extension of a variable of type \texttt{char} on widening to integer is implementation-defined.

The \texttt{ferror()} or \texttt{feof()} functions must be used to distinguish between an error condition and an end-of-file condition.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

\texttt{feof()}, \texttt{ferror()}, \texttt{fopen()}, \texttt{getchar()}, \texttt{getc()}, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY

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

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are marked.

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

- The [EIO] and [EOVERFLOW] mandatory error conditions are added.
- The [ENOMEM] and [ENXIO] optional error conditions are added.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The DESCRIPTION is updated to clarify the behavior when the end-of-file indicator for the input stream is not set.
- The RETURN VALUE section is updated to note that the error indicator shall be set for the stream.
NAME
fgetpos — get current file position information

SYNOPSIS
#include <stdio.h>

int fgetpos(FILE *restrict stream, fpos_t *restrict pos);

DESCRIPTION
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

The fgetpos() function shall store the current values of the parse state (if any) and file position
indicator for the stream pointed to by stream in the object pointed to by pos. The value stored
contains unspecified information usable by fsetpos() for repositioning the stream to its position
at the time of the call to fgetpos().

RETURN VALUE
Upon successful completion, fgetpos() shall return 0; otherwise, it shall return a non-zero value
and set errno to indicate the error.

ERRORS
The fgetpos() function shall fail if:

CX [EOVERFLOW] The current value of the file position cannot be represented correctly in an
object of type fpos_t.

The fgetpos() function may fail if:

CX [EBADF] The file descriptor underlying stream is not valid.

CX [ESPIPE] The file descriptor underlying stream is associated with a pipe, FIFO, or socket.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fopen(), ftell(), rewind(), ungetc(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 4. Derived from the ISO C standard.

Issue 5
Large File Summit extensions are added.

Issue 6
Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the
Single UNIX Specification:
• The [EBADF] and [ESPIPE] optional error conditions are added.

An additional [ESPIPE] error condition is added for sockets.

The prototype for \texttt{fgetpos()} is changed for alignment with the ISO/IEC 9899:1999 standard.
NAME
fgets — get a string from a stream

SYNOPSIS
#include <stdio.h>
char *fgets(char *restrict s, int n, FILE *restrict stream);

DESCRIPTION
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
The fgets() function shall read bytes from stream into the array pointed to by s, until n−1 bytes
are read, or a <newline> is read and transferred to s, or an end-of-file condition is encountered.
The string is then terminated with a null byte.
The fgets() 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(),
fgetswc(), fgetsws(), fread(), fscanf(), getc(), getchar(), gets(), or scanf() using stream that returns
data not supplied by a prior call to ungetc() or ungetwc().

RETURN VALUE
Upon successful completion, fgets() shall return s. If the stream is at end-of-file, the end-of-file
indicator for the stream shall be set and fgets() shall return a null pointer. If a read error occurs,
the error indicator for the stream shall be set, fgets() shall return a null pointer, and shall set
errno to indicate the error.

ERRORS
Refer to fgetc().

EXAMPLES

Reading Input
The following example uses fgets() to read each line of input. {LINE_MAX}, which defines the
maximum size of the input line, is defined in the <limits.h> header.

#include <stdio.h>
...
char line[LINE_MAX];
...
while (fgets(line, LINE_MAX, fp) != NULL) {
...
}

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO

fopen(), fread(), gets(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY

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

Extensions beyond the ISO C standard are marked.

The prototype for fgets() is changed for alignment with the ISO/IEC 9899: 1999 standard.
NAME
fgetwc — get a wide-character code from a stream

SYNOPSIS
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *stream);

DESCRIPTION
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 fgetwc() function shall obtain the next character (if present) from the input stream pointed to by stream, convert that to the corresponding wide-character code, and advance the associated file position indicator for the stream (if defined).

If an error occurs, the resulting value of the file position indicator for the stream is unspecified.

The fgetwc() 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(), gets(), getchar(), gets(), or scanf() using stream that returns data not supplied by a prior call to ungetc() or ungetwc().

RETURN VALUE
Upon successful completion, the fgetwc() function shall return the wide-character code of the character read from the input stream pointed to by stream converted to a type wint_t. If the stream is at end-of-file, the end-of-file indicator for the stream shall be set and fgetwc() shall return WEOF. If a read error occurs, the error indicator for the stream shall be set, fgetwc() shall return WEOF, and shall set errno to indicate the error. If an encoding error occurs, the error indicator for the stream shall be set, fgetwc() shall return WEOF, and shall set errno to indicate the error.

ERRORS
The fgetwc() function shall fail if data needs to be read and:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying stream and the process would be delayed in the fgetwc() operation.

[EBADF] The file descriptor underlying stream is not a valid file descriptor open for reading.

[EILSEQ] The data obtained from the input stream does not form a valid character.

[EINTR] The read operation was terminated due to the receipt of a signal, and no data was transferred.

[EIO] A physical I/O error has occurred, or the process is in a background process group attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTOIN signal or the process group is orphaned.

This error may also be generated for implementation-defined reasons.

[EOVERFLOW] The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

The fgetwc() function may fail if:

[ENOMEM] Insufficient storage space is available.
A request was made of a nonexistent device, or the request was outside the capabilities of the device.

**EXAMPLES**
None.

**APPLICATION USAGE**
The `ferror()` or `feof()` functions must be used to distinguish between an error condition and an end-of-file condition.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`feof()`, `ferror()`, `fopen()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`, `<wchar.h>`

**CHANGE HISTORY**
First released in Issue 4. Derived from the MSE working draft.

**Issue 5**
The Optional Header (OH) marking is removed from `<stdio.h>`.

**Issue 6**
Extensions beyond the ISO C standard are marked.

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

- The [EIO] and [EOVERFLOW] mandatory error conditions are added.
- The [ENOMEM] and [ENXIO] optional error conditions are added.
NAME
fgetws — get a wide-character string from a stream

SYNOPSIS
#include <stdio.h>
#include <wchar.h>

wchar_t *fgetws(wchar_t *restrict ws, int n, FILE *restrict stream);

DESCRIPTION
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.

If an error occurs, the resulting value of the file position indicator for the stream is unspecified.

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().

RETURN VALUE
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.

ERRORS
Refer to fgetwc().

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fopen(), fread(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.

Issue 5
The Optional Header (OH) marking is removed from <stdio.h>.
11979 **Issue 6**

11980 Extensions beyond the ISO C standard are marked.

11981 The prototype for `fgetws()` is changed for alignment with the ISO/IEC 9899:1999 standard.
NAME
fileno — map a stream pointer to a file descriptor

SYNOPSIS
CX
#include <stdio.h>

int fileno(FILE *stream);

DESCRIPTION
The fileno() function shall return the integer file descriptor associated with the stream pointed to by stream.

RETURN VALUE
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.

ERRORS
The fileno() function may fail if:

[EBADF] The stream argument is not a valid stream.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
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).

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.5.1 (on page 35), fdopen(), fopen(), stdin, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

• The [EBADF] optional error condition is added.
NAME
flockfile, ftrylockfile, funlockfile — stdio locking functions

SYNOPSIS
#include <stdio.h>

void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);

DESCRIPTION
These functions shall provide for explicit application-level locking of stdio (FILE *) objects.
These functions can be used by a thread to delineate a sequence of I/O statements that are
executed as a unit.
The flockfile() function shall acquire for a thread ownership of a (FILE *) object.
The ftrylockfile() function shall acquire for a thread ownership of a (FILE *) object if the object is
available; ftrylockfile() is a non-blocking version of flockfile().
The funlockfile() function shall relinquish the ownership granted to the thread. The behavior is
undefined if a thread other than the current owner calls the funlockfile() function.
The functions shall behave as if there is a lock count associated with each (FILE *) object. This
count is implicitly initialized to zero when the (FILE *) object is created. The (FILE *) object is
unlocked when the count is zero. When the count is positive, a single thread owns the (FILE *)
object. When the flockfile() function is called, if the count is zero or if the count is positive and
the caller owns the (FILE *) object, the count shall be incremented. Otherwise, the calling thread
shall be suspended, waiting for the count to return to zero. Each call to funlockfile() shall
decrement the count. This allows matching calls to flockfile() (or successful calls to ftrylockfile())
and funlockfile() to be nested.
All functions that reference (FILE *) objects shall behave as if they use flockfile() and funlockfile()
internally to obtain ownership of these (FILE *) objects.

RETURN VALUE
None for flockfile() and funlockfile().
The ftrylockfile() function shall return zero for success and non-zero to indicate that the lock
cannot be acquired.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
Applications using these functions may be subject to priority inversion, as discussed in the Base

RATIONALE
The flockfile() and funlockfile() functions provide an orthogonal mutual-exclusion lock for each
FILE. The ftrylockfile() function provides a non-blocking attempt to acquire a file lock,
analogous to pthread_mutex_trylock().
These locks behave as if they are the same as those used internally by stdio for thread-safety.
This both provides thread-safety of these functions without requiring a second level of internal
locking and allows functions in stdio to be implemented in terms of other stdio functions.
Application writers and implementors should be aware that there are potential deadlock problems on `FILE` objects. For example, the line-buffered flushing semantics of `stdio` (requested via `_IOLBF`) require that certain input operations sometimes cause the buffered contents of implementation-defined line-buffered output streams to be flushed. If two threads each hold the lock on the other's `FILE`, deadlock ensues. This type of deadlock can be avoided by acquiring `FILE` locks in a consistent order. In particular, the line-buffered output stream deadlock can typically be avoided by acquiring locks on input streams before locks on output streams if a thread would be acquiring both.

In summary, threads sharing `stdio` streams with other threads can use `flockfile()` and `funlockfile()` to cause sequences of I/O performed by a single thread to be kept bundled. The only case where the use of `flockfile()` and `funlockfile()` is required is to provide a scope protecting uses of the `*_unlocked()` functions/macros. This moves the cost/performance tradeoff to the optimal point.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`getc_unlocked()`, `putc_unlocked()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`

**CHANGE HISTORY**

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

**Issue 6**

These functions are marked as part of the Thread-Safe Functions option.
NAME
floor, floorf, floorl — floor function

SYNOPSIS
#include <math.h>

double floor(double x);
float floorf(float x);
long double floorl(long double x);

DESCRIPTION
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.

These functions shall compute the largest integral value not greater than \( x \).

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

RETURN VALUE
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.

If \( x \) is NaN, a NaN shall be returned.

If \( x \) is ±0 or ±Inf, \( x \) shall be returned.

If the correct value would cause overflow, a range error shall occur and floor(), floorf(), and floorl() shall return the value of the macro \(-\text{HUGE}_\text{VAL}, -\text{HUGE}_\text{VALF}, \text{and} -\text{HUGE}_\text{VALL}\), respectively.

ERRORS
These functions shall fail if:

Range Error The result would cause an overflow.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then \( \text{errno} \) 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.

EXAMPLES
None.

APPLICATION USAGE
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.

The floor() function can only overflow when the floating-point representation has DBL_MANT_DIG > DBL_MAX_EXP.

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

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
ceil(), 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>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The floor() and floorl() functions are added for alignment with the ISO/IEC 9899:1999 standard.

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

NAME
fma, fmaf, fmal — floating-point multiply-add

SYNOPSIS
#include <math.h>

double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);

DESCRIPTION
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.

These functions shall compute \((x \times y) + z\), 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.

An application wishing to check for error situations should set \texttt{errno} to zero and call \texttt{fetestexcept}(FE_ALL_EXCEPT) before calling these functions. On return, if \texttt{errno} is non-zero or \texttt{fetestexcept}(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE
Upon successful completion, these functions shall return \((x \times y) + z\), rounded as one ternary operation.

If \(x\) or \(y\) are NaN, a NaN shall be returned.

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.

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.

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.

If \(x\times y\) is not 0*Inf nor Inf*0 and \(z\) is a NaN, a NaN shall be returned.

ERRORS
These functions shall fail if:

Domain Error The value of \(x\times y+z\) is invalid, or the value \(x\times y\) is invalid and \(z\) is not a NaN.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then \texttt{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.

Range Error The result overflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then \texttt{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.

These functions may fail if:
Domain Error The value $x\cdot y$ is invalid and $z$ is a NaN.

If the integer expression (math_errno & MATH_ERRNO) is non-zero, then $errno$ shall be set to [EDOM]. If the integer expression (math_errno & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

Range Error The result underflows.

If the integer expression (math_errno & MATH_ERRNO) is non-zero, then $errno$ shall be set to [ERANGE]. If the integer expression (math_errno & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXEMPLARY
None.

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

RATIONALE
In many cases, clever use of floating (fused) multiply-add leads to much improved code; but its unexpected use by the compiler can undermine carefully written code. The FP_CONTRACT macro can be used to disallow use of floating multiply-add; and the fma() function guarantees its use where desired. Many current machines provide hardware floating multiply-add instructions; software implementation can be used for others.

FUTURE DIRECTIONS
None.

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

CHANGE HISTORY
NAME
fmax, fmaxf, fmaxl — determine maximum numeric value of two floating-point numbers

SYNOPSIS
#include <math.h>

double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);

DESCRIPTION
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.

These functions shall determine the maximum numeric value of their arguments. NaN arguments shall be treated as missing data: if one argument is a NaN and the other numeric, then these functions shall choose the numeric value.

RETURN VALUE
Upon successful completion, these functions shall return the maximum numeric value of their arguments.

If just one argument is a NaN, the other argument shall be returned.

If x and y are NaN, a NaN shall be returned.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fdim(), fmin(), the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

CHANGE HISTORY
NAME
fmin, fminf, fminl — determine minimum numeric value of two floating-point numbers

SYNOPSIS
#include <math.h>

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);

DESCRIPTION
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.

These functions shall determine the minimum numeric value of their arguments. NaN arguments shall be treated as missing data: if one argument is a NaN and the other numeric, then these functions shall choose the numeric value.

RETURN VALUE
Upon successful completion, these functions shall return the minimum numeric value of their arguments.
If just one argument is a NaN, the other argument shall be returned.
If x and y are NaN, a NaN shall be returned.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fdim(), fmax(), the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

CHANGE HISTORY
NAME
fmod, fmodf, fmodl — floating-point remainder value function

SYNOPSIS
#include <math.h>

double fmod(double x, double y);
fmod ( )

g float fmodf(float x, float y);

long double fmodl(long double x, long double y);

DESCRIPTION
The functionality described on this reference page is aligned with the ISO C standard. An
collection between the requirements described here and the ISO C standard is unintentional. This

These functions shall return the floating-point remainder of the division of x by y.

An application wishing to check for error situations should set errno to zero and call
fclearexcept(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.

RETURN VALUE
These functions shall return the value \(x - i \times y\), 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 \(y\).

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.

If \(x\) or \(y\) is NaN, a NaN shall be returned.

If \(y\) is zero, a domain error shall occur, and either a NaN (if supported), or an implementation-defined
value shall be returned.

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

If \(x\) is \pm 0 and \(y\) is not zero, \pm 0 shall be returned.

If \(x\) is not infinite and \(y\) is \pm \text{Inf}, \(x\) shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and
the correct value shall be returned.

ERRORS
These functions shall fail if:

Domain Error The \(x\) argument is infinite or \(y\) is zero.

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.

These functions may fail if:

Range Error The result underflows.

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.
EXCEPTIONS
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fselectexcept(), fetestexcept(), isnan(), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The behavior for when the y argument is zero is now defined.

The fmodf() and fmodl() functions are added for alignment with the ISO/IEC 9899:1999 standard.

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

NAME
fmtmsg — display a message in the specified format on standard error and/or a system console

SYNOPSIS
XSI
#include <fmtmsg.h>

int fmtmsg(long classification, const char *label, int severity,
const char *text, const char *action, const char *tag);

DESCRIPTION
The fmtmsg() function shall display messages in a specified format instead of the traditional printf() function.

Based on a message's classification component, fmtmsg() shall write a formatted message either to standard error, to the console, or to both.

A formatted message consists of up to five components as defined below. The component classification is not part of a message displayed to the user, but defines the source of the message and directs the display of the formatted message.

classification Contains the sum of identifying values constructed from the constants defined below. Any one identifier from a subclass may be used in combination with a single identifier from a different subclass. Two or more identifiers from the same subclass should not be used together, with the exception of identifiers from the display subclass. (Both display subclass identifiers may be used so that messages can be displayed to both standard error and the system console.)

Major Classifications
Identifies the source of the condition. Identifiers are: MM_HARD (hardware), MM_SOFT (software), and MM_FIRM (firmware).

Message Source Subclassifications
Identifies the type of software in which the problem is detected. Identifiers are: MM_APPL (application), MM_UTIL (utility), and MM_OPSYS (operating system).

Display Subclassifications
Indicates where the message is to be displayed. Identifiers are: MM_PRINT to display the message on the standard error stream, MM_CONSOLE to display the message on the system console. One or both identifiers may be used.

Status Subclassifications
Indicates whether the application can recover from the condition. Identifiers are: MM_RECOVER (recoverable) and MM_NRECOV (non-recoverable).

An additional identifier, MM_NULLMC, indicates that no classification component is supplied for the message.

label Identifies the source of the message. The format is two fields separated by a colon. The first field is up to 10 bytes, the second is up to 14 bytes.

severity Indicates the seriousness of the condition. Identifiers for the levels of severity are:
System Interfaces

fmtmsg()

12392 MM_HALT Indicates that the application has encountered a severe fault and is halting. Produces the string "HALT".
12393
12394 MM_ERROR Indicates that the application has detected a fault. Produces the string "ERROR".
12395
12396 MM_WARNING Indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the string "WARNING".
12397
12398 MM_INFO Provides information about a condition that is not in error. Produces the string "INFO".
12399
12400 MM_NOSEV Indicates that no severity level is supplied for the message.
12401
text 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.
12402
12403 action Describes the first step to be taken in the error-recovery process. The fmtmsg() function precedes the action string with the prefix: "TO FIX: " The action string is not limited to a specific size.
12404
12405 tag An identifier that references on-line documentation for the message. Suggested usage is that tag includes the label and a unique identifying number. A sample tag is "XSI:cat:146".
12406
12407 The MSGVERB environment variable (for message verbosity) shall determine for fmtmsg() which message components it is to select when writing messages to standard error. The value of MSGVERB shall be a colon-separated list of optional keywords. Valid keywords are: label, severity, text, action, and tag. If MSGVERB contains a keyword for a component and the component’s value is not the component’s null value, fmtmsg() shall include that component in the message when writing the message to standard error. If MSGVERB 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 MSGVERB 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, fmtmsg() shall select all components.
12408
12409 MSGVERB shall determine which components are selected for display to standard error. All message components shall be included in console messages.
12410
12411 RETURN VALUE
12412 The fmtmsg() function shall return one of the following values:
12413 MM_OK The function succeeded.
12414 MM_NOTOK The function failed completely.
12415 MM_NOMSG The function was unable to generate a message on standard error, but otherwise succeeded.
12416 MM_NOCON The function was unable to generate a console message, but otherwise succeeded.
12417
12418 ERRORS
12419 None.
EXAMPLES

1. The following example of `fmtmsg()`:

   ```c
   fmtmsg(MM_PRINT, "XSI:cat", MM_ERROR, "illegal option",
         "refer to cat in user’s reference manual", "XSI:cat:001")
   ```

   produces a complete message in the specified message format:

   ```
   XSI:cat: ERROR: illegal option
   TO FIX: refer to cat in user’s reference manual XSI:cat:001
   ```

2. When the environment variable `MSGVERB` is set as follows:

   ```
   MSGVERB=severity:text:action
   ```

   and Example 1 is used, `fmtmsg()` produces:

   ```
   ERROR: illegal option
   TO FIX: refer to cat in user’s reference manual
   ```

APPLICATION USAGE

One or more message components may be systematically omitted from messages generated by
an application by using the null value of the argument for that component.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`printf()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<fmtmsg.h>`

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.
NAME

fnmatch — match a filename or a pathname

SYNOPSIS

#include <fnmatch.h>

int fnmatch(const char *pattern, const char *string, int flags);

DESCRIPTION

The fnmatch() function shall match patterns as described in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.13.1, Patterns Matching a Single Character, and Section 2.13.2, Patterns Matching Multiple Characters. It checks the string specified by the string argument to see if it matches the pattern specified by the pattern argument.

The flags argument shall modify the interpretation of pattern and string. It is the bitwise-inclusive OR of zero or more of the flags defined in <fnmatch.h>. If the FNM_PATHNAME flag is set in flags, then a slash character (’/’) in string shall be explicitly matched by a slash in pattern; it shall not be matched by either the asterisk or question-mark special characters, nor by a bracket expression. If the FNM_PATHNAME flag is not set, the slash character shall be treated as an ordinary character.

If FNM_NOESCAPE is not set in flags, a backslash character (’\’) in pattern followed by any other character shall match that second character in string. In particular, "\" shall match a backslash in string. If FNM_NOESCAPE is set, a backslash character shall be treated as an ordinary character.

If FNM_PERIOD is set in flags, then a leading period (’.’) in string shall match a period in pattern; as described by rule 2 in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.13.3, Patterns Used for Filename Expansion where the location of “leading” is indicated by the value of FNM_PATHNAME:

- If FNM_PATHNAME is set, a period is “leading” if it is the first character in string or if it immediately follows a slash.
- If FNM_PATHNAME is not set, a period is “leading” only if it is the first character of string.

If FNM_PERIOD is not set, then no special restrictions are placed on matching a period.

RETURN VALUE

If string matches the pattern specified by pattern, then fnmatch() shall return 0. If there is no match, fnmatch() shall return FNM_NOMATCH, which is defined in <fnmatch.h>. If an error occurs, fnmatch() shall return another non-zero value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The fnmatch() function has two major uses. It could be used by an application or utility that needs to read a directory and apply a pattern against each entry. The find utility is an example of this. It can also be used by the pax utility to process its pattern operands, or by applications that need to match strings in a similar manner.

The name fnmatch() is intended to imply filename match, rather than pathname match. The default action of this function is to match filenames, rather than pathnames, since it gives no special significance to the slash character. With the FNM_PATHNAME flag, fnmatch() does match pathnames, but without tilde expansion, parameter expansion, or special treatment for a period.
fnmatch()

at the beginning of a filename.

RATIONALITY
This function replaced the REG_FILENAME flag of regcomp() in early proposals of this volume of IEEE Std 1003.1-2001. It provides virtually the same functionality as the regcomp() and regexec() functions using the REG_FILENAME and REG_FSLASH flags (the REG_FSLASH flag was proposed for regcomp(), and would have had the opposite effect from FNM_PATHNAME), but with a simpler function and less system overhead.

FUTURE DIRECTIONS
None.

SEE ALSO
glob(), wordexp(), the Base Definitions volume of IEEE Std 1003.1-2001, <fnmatch.h>, the Shell and Utilities volume of IEEE Std 1003.1-2001

CHANGE HISTORY

Issue 5
Moved from POSIX2 C-language Binding to BASE.
NAME
fopen — open a stream

SYNOPSIS
#include <stdio.h>
FILE *fopen(const char *restrict filename, const char *restrict mode);

DESCRIPTION
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
The fopen() function shall open the file whose pathname is the string pointed to by filename, and
associates a stream with it.
The mode argument points to a string. If the string is one of the following, the file shall be opened
in the indicated mode. Otherwise, the behavior is undefined.
r or rb
Open file for reading.
w or wb
Truncate to zero length or create file for writing.
a or ab
Append; open or create file for writing at end-of-file.
r+ or rb+ or r+b
Open file for update (reading and writing).
w+ or wb+ or w+b
Truncate to zero length or create file for update.
a+ or ab+ or a+b
Append; open or create file for update, writing at end-of-file.
The character ‘b’ shall have no effect, but is allowed for ISO C standard conformance. Opening
a file with read mode (r as the first character in the mode argument) shall fail if the file does not
exist or cannot be read.
Opening a file with append mode (a as the first character in the mode argument) shall cause all
subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening
calls to fseek().
When a file is opened with update mode (‘+’ as the second or third character in the mode
argument), both input and output may be performed on the associated stream. However, the
application shall ensure that output is not directly followed by input without an intervening call
to flush() or to a file positioning function (fseek(), fsetpos(), or rewind()), and input is not directly
followed by output without an intervening call to a file positioning function, unless the input
operation encounters end-of-file.
When opened, a stream is fully buffered if and only if it can be determined not to refer to an
interactive device. The error and end-of-file indicators for the stream shall be cleared.
If mode is w, wb, a, ab, w+, wb+, w+b, a+, ab+, or a+b, and the file did not previously exist, upon
successful completion, the fopen() function shall mark for update the st_atime, st_ctime, and
st_mtime fields of the file and the st_ctime and st_mtime fields of the parent directory.
If mode is w, wb, w+, wb+, or w+b, and the file did previously exist, upon successful completion,
fopen() shall mark for update the st_ctime and st_mtime fields of the file. The fopen() function
shall allocate a file descriptor as open() does.
After a successful call to the fopen() function, the orientation of the stream shall be cleared, the
encoding rule shall be cleared, and the associated mbstate_t object shall be set to describe an
initial conversion state.
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.

**RETURN VALUE**

Upon successful completion, `fopen()` shall return a pointer to the object controlling the stream. Otherwise, a null pointer shall be returned, and `errno` shall be set to indicate the error.

**ERRORS**

The `fopen()` function shall fail if:

- **[EACCES]** Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by `mode` are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
- **[EINTR]** A signal was caught during `fopen()`.
- **[EISDIR]** The named file is a directory and `mode` requires write access.
- **[ELOOP]** A loop exists in symbolic links encountered during resolution of the `path` argument.
- **[EMFILE]** `OPEN_MAX` file descriptors are currently open in the calling process.
- **[ENAMETOOLONG]** The length of the `filename` argument exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.
- **[ENFILE]** The maximum allowable number of files is currently open in the system.
- **[ENOENT]** A component of `filename` does not name an existing file or `filename` is an empty string.
- **[ENOSPC]** The directory or file system that would contain the new file cannot be expanded, the file does not exist, and the file was to be created.
- **[ENOTDIR]** A component of the path prefix is not a directory.
- **[ENXIO]** The named file is a character special or block special file, and the device associated with this special file does not exist.
- **[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`.
- **[EROFS]** The named file resides on a read-only file system and `mode` requires write access.
- The `fopen()` function may fail if:
  - **[EINVAL]** The value of the `mode` argument is not valid.
  - **[ELOOP]** More than `SYMLOOP_MAX` symbolic links were encountered during resolution of the `path` argument.
  - **[EMFILE]** `FOPEN_MAX` streams are currently open in the calling process.
  - **[EMFILE]** `STREAM_MAX` streams are currently open in the calling process.
  - **[ENAMETOOLONG]** Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`. 
fopen()

**System Interfaces**

12601 CX  [ENOMEM] Insufficient storage space is available.

12602 CX  [ETXTBSY] The file is a pure procedure (shared text) file that is being executed and mode requires write access.

12604 **EXAMPLES**

12605 **Opening a File**

12606 The following example tries to open the file named file for reading. The fopen() function returns a file pointer that is used in subsequent fgets() and fclose() calls. If the program cannot open the file, it just ignores it.

```c
#include <stdio.h>
...
FILE *fp;
...
void rgrep(const char *file)
{
...
  if ((fp = fopen(file, "r")) == NULL)
    return;
...
}
```

**APPLICATION USAGE**

None.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

fclose(), fdopen(), freopen(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

**CHANGE HISTORY**

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

**Issue 5**

Large File Summit extensions are added.

**Issue 6**

Extensions beyond the ISO C standard are marked.

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

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.
- The [ELOOP] mandatory error condition is added.
- The [EINVAL], [EMFILE], [ENAMETOOLONG], [ENOMEM], and [ETXTBSY] optional error conditions are added.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The prototype for `fopen()` is updated.
- The DESCRIPTION is updated to note that if the argument `mode` points to a string other than those listed, then the behavior is undefined.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.
fork() — create a new process

#include <unistd.h>

pid_t fork(void);

The fork() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except as detailed below:

- The child process shall have a unique process ID.
- The child process ID also shall not match any active process group ID.
- The child process shall have a different parent process ID, which shall be the process ID of the calling process.
- The child process shall have its own copy of the parent’s file descriptors. Each of the child’s file descriptors shall refer to the same open file description with the corresponding file descriptor of the parent.
- The child process shall have its own copy of the parent’s open directory streams. Each open directory stream in the child process may share directory stream positioning with the corresponding directory stream of the parent.
- The child process shall have its own copy of the parent’s message catalog descriptors.
- All semadj values shall be cleared.
- File locks set by the parent process shall not be inherited by the child process.
- The set of signals pending for the child process shall be initialized to the empty set.
- Interval timers shall be reset in the child process.
- Any semaphores that are open in the parent process shall also be open in the child process.
- The child process shall not inherit any address space memory locks established by the parent process via calls to mlockall() or mlock().
- Memory mappings created in the parent shall be retained in the child process. MAP_PRIVATE mappings inherited from the parent shall also be MAP_PRIVATE mappings in the child, and any modifications to the data in these mappings made by the parent prior to calling fork() shall be visible to the child. Any modifications to the data in MAP_PRIVATE 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.
- For the SCHED_FIFO and SCHED_RR scheduling policies, the child process shall inherit the 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.
- Per-process timers created by the parent shall not be inherited by the child process.
- 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...
fork( )

• No asynchronous input or asynchronous output operations shall be inherited by the child process.

• A process shall be created with a single thread. If a multi-threaded process calls fork(), 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 exec functions is called. Fork handlers may be established by means of the pthread_atfork() function in order to maintain application invariants across fork() calls.

TRC TRI
If the Trace option and the Trace Inherit option are both supported:

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 posix_trace_attr_setinherited() function.

TRC
If the Trace option is supported, but the Trace Inherit option is not supported:

The child process shall not be traced into any of the trace streams of its parent process.

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.

CPT
The initial value of the CPU-time clock of the child process shall be set to zero.

TCT
The initial value of the CPU-time clock of the single thread of the child process shall be set to zero.

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.

After fork(), both the parent and the child processes shall be capable of executing independently before either one terminates.

RETURN VALUE

Upon successful completion, fork() 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 fork() function. Otherwise, −1 shall be returned to the parent process, no child process shall be created, and errno shall be set to indicate the error.

ERRORS

The fork() function shall fail if:

[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.

[ENOMEM] Insufficient storage space is available.
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
Many historical implementations have timing windows where a signal sent to a process group (for example, an interactive SIGINT) just prior to or during execution of fork() is delivered to the parent following the fork() but not to the child because the fork() code clears the child’s set of pending signals. This volume of IEEE Std 1003.1-2001 does not require, or even permit, this behavior. However, it is pragmatic to expect that problems of this nature may continue to exist in implementations that appear to conform to this volume of IEEE Std 1003.1-2001 and pass available verification suites. This behavior is only a consequence of the implementation failing to make the interval between signal generation and delivery totally invisible. From the application’s perspective, a fork() call should appear atomic. A signal that is generated prior to the fork() should be delivered prior to the fork(). A signal sent to the process group after the fork() should be delivered to both parent and child. The implementation may actually initialize internal data structures corresponding to the child’s set of pending signals to include signals sent to the process group during the fork(). Since the fork() call can be considered as atomic from the application’s perspective, the set would be initialized as empty and such signals would have arrived after the fork(); see also <signal.h>.

One approach that has been suggested to address the problem of signal inheritance across fork() is to add an [EINTR] error, which would be returned when a signal is detected during the call. While this is preferable to losing signals, it was not considered an optimal solution. Although it is not recommended for this purpose, such an error would be an allowable extension for an implementation.

The [ENOMEM] error value is reserved for those implementations that detect and distinguish such a condition. This condition occurs when an implementation detects that there is not enough memory to create the process. This is intended to be returned when [EAGAIN] is inappropriate because there can never be enough memory (either primary or secondary storage) to perform the operation. Since fork() duplicates an existing process, this must be a condition where there is sufficient memory for one such process, but not for two. Many historical implementations actually return [ENOMEM] due to temporary lack of memory, a case that is not generally distinct from [EAGAIN] from the perspective of a conforming application.

Part of the reason for including the optional error [ENOMEM] is because the SVID specifies it and it should be reserved for the error condition specified there. The condition is not applicable on many implementations.

IEEE Std 1003.1-1988 neglected to require concurrent execution of the parent and child of fork(). A system that single-threads processes was clearly not intended and is considered an unacceptable “toy implementation” of this volume of IEEE Std 1003.1-2001. The only objection anticipated to the phrase “executing independently” is testability, but this assertion should be testable. Such tests require that both the parent and child can block on a detectable action of the other, such as a write to a pipe or a signal. An interactive exchange of such actions should be possible for the system to conform to the intent of this volume of IEEE Std 1003.1-2001.

The [EAGAIN] error exists to warn applications that such a condition might occur. Whether it occurs or not is not in any practical sense under the control of the application because the condition is usually a consequence of the user’s use of the system, not of the application’s code. Thus, no application can or should rely upon its occurrence under any circumstances, nor should the exact semantics of what concept of “user” is used be of concern to the application writer. Validation writers should be cognizant of this limitation.
There are two reasons why POSIX programmers call `fork()`. One reason is to create a new thread of control within the same program (which was originally only possible in POSIX by creating a new process); the other is to create a new process running a different program. In the latter case, the call to `fork()` is soon followed by a call to one of the `exec` functions.

The general problem with making `fork()` work in a multi-threaded world is what to do with all of the threads. There are two alternatives. One is to copy all of the threads into the new process. This causes the programmer or implementation to deal with threads that are suspended on system calls or that might be about to execute system calls that should not be executed in the new process. The other alternative is to copy only the thread that calls `fork()`. This creates the difficulty that the state of process-local resources is usually held in process memory. If a thread that is not calling `fork()` holds a resource, that resource is never released in the child process because the thread whose job it is to release the resource does not exist in the child process.

When a programmer is writing a multi-threaded program, the first described use of `fork()`, creating new threads in the same program, is provided by the `pthread_create()` function. The `fork()` function is thus used only to run new programs, and the effects of calling functions that require certain resources between the call to `fork()` and the call to an `exec` function are undefined.

The addition of the `forkall()` function to the standard was considered and rejected. The `forkall()` function lets all the threads in the parent be duplicated in the child. This essentially duplicates the state of the parent in the child. This allows threads in the child to continue processing and allows locks and the state to be preserved without explicit `pthread_atfork()` code. The calling process has to ensure that the threads processing state that is shared between the parent and child (that is, file descriptors or MAP_SHARED memory) behaves properly after `forkall()`.

For example, if a thread is reading a file descriptor in the parent when `forkall()` is called, then two threads (one in the parent and one in the child) are reading the file descriptor after the `forkall()`.

If this is not desired behavior, the parent process has to synchronize with such threads before calling `forkall()`.

When `forkall()` is called, threads, other than the calling thread, that are in functions that can return with an [EINTR] error may have those functions return [EINTR] if the implementation cannot ensure that the function behaves correctly in the parent and child. In particular, `pthread_cond_wait()` and `pthread_cond_timedwait()` need to return in order to ensure that the condition has not changed. These functions can be awakened by a spurious condition wakeup rather than returning [EINTR].

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`alarm()`, `exec`, `fcntl()`, `posix_trace_attr_getinherited()`, `posix_trace_trid_eventid_open()`, `semop()`, `signal()`, `times()`.

**CHANGE HISTORY**

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

**Issue 5**

The DESCRIPTION is changed for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

**Issue 6**

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

- The effect of `fork()` on a pending alarm call in the child process is clarified.
- The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.
- The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.
**NAME**

fpathconf, pathconf — get configurable pathname variables

**SYNOPSIS**

```c
#include <unistd.h>

long fpathconf(int fildes, int name);
long pathconf(const char *path, int name);
```

**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.

For `pathconf()`, the `path` argument points to the pathname of a file or directory.

For `fpathconf()`, the `fildes` argument is an open file descriptor.

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`. Support for some pathname configuration variables is dependent on implementation options (see shading and margin codes in the table below). Where an implementation option is not supported, the variable need not be supported.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value of <code>name</code></th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{FILESIZEBITS}</code></td>
<td>_PC_FILESIZEBITS</td>
<td>3, 4</td>
</tr>
<tr>
<td><code>{LINK_MAX}</code></td>
<td>_PC_LINK_MAX</td>
<td>1</td>
</tr>
<tr>
<td><code>{MAX_CANON}</code></td>
<td>_PC_MAX_CANON</td>
<td>2</td>
</tr>
<tr>
<td><code>{MAX_INPUT}</code></td>
<td>_PC_MAX_INPUT</td>
<td>2</td>
</tr>
<tr>
<td><code>{NAME_MAX}</code></td>
<td>_PC_NAME_MAX</td>
<td>3, 4</td>
</tr>
<tr>
<td><code>{PATH_MAX}</code></td>
<td>_PC_PATH_MAX</td>
<td>4, 5</td>
</tr>
<tr>
<td><code>{PIPE_BUF}</code></td>
<td>_PC_PIPE_BUF</td>
<td>6</td>
</tr>
<tr>
<td><code>{POSIX_ALLOC_SIZE_MIN}</code></td>
<td>_PC_ALLOC_SIZE_MIN</td>
<td></td>
</tr>
<tr>
<td><code>{POSIX_REC_INCR_XFER_SIZE}</code></td>
<td>_PC_REC_INCR_XFER_SIZE</td>
<td></td>
</tr>
<tr>
<td><code>{POSIX_REC_MAX_XFER_SIZE}</code></td>
<td>_PC_REC_MAX_XFER_SIZE</td>
<td></td>
</tr>
<tr>
<td><code>{POSIX_REC_MIN_XFER_SIZE}</code></td>
<td>_PC_REC_MIN_XFER_SIZE</td>
<td></td>
</tr>
<tr>
<td><code>{POSIX_REC_XFER_ALIGN}</code></td>
<td>_PC_REC_XFER_ALIGN</td>
<td></td>
</tr>
<tr>
<td><code>{SYMLINK_MAX}</code></td>
<td>_PC_SYMLINK_MAX</td>
<td>4, 9</td>
</tr>
<tr>
<td><code>{POSIX_CHOWN_RESTRICTED}</code></td>
<td>_PC_CHOWN_RESTRICTED</td>
<td>7</td>
</tr>
<tr>
<td><code>{POSIX_NO_TRUNC}</code></td>
<td>_PC_NO_TRUNC</td>
<td>3, 4</td>
</tr>
<tr>
<td><code>{POSIX_VDISABLE}</code></td>
<td>_PC_VDISABLE</td>
<td>2</td>
</tr>
<tr>
<td><code>{POSIX_ASYNC_IO}</code></td>
<td>_PC_ASYNC_IO</td>
<td>8</td>
</tr>
<tr>
<td><code>{POSIX_PRIO_IO}</code></td>
<td>_PC_PRIO_IO</td>
<td>8</td>
</tr>
<tr>
<td><code>{POSIX_SYNC_IO}</code></td>
<td>_PC_SYNC_IO</td>
<td>8</td>
</tr>
</tbody>
</table>
**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.

**RETURN VALUE**

If `name` is an invalid value, both `pathconf()` and `fpathconf()` shall return -1 and set `errno` to indicate the error.

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>`.

**ERRORS**

The `pathconf()` function shall fail if:

- `[EINVAL]` The value of `name` is not valid.
- `[ELOOP]` A loop exists in symbolic links encountered during resolution of the `path` argument.

The `pathconf()` function may fail if:
The `fpathconf()` function was proposed immediately after the `sysconf()` function when it was realized that some configurable values may differ across file system, directory, or device boundaries.

For example, `{NAME_MAX}` frequently changes between System V and BSD-based file systems; System V uses a maximum of 14, BSD 255. On an implementation that provides both types of file systems, an application would be forced to limit all pathname components to 14 bytes, as this would be the value specified in `<limits.h>` on such a system.

Therefore, various useful values can be queried on any pathname or file descriptor, assuming that the appropriate permissions are in place.

The value returned for the variable `{PATH_MAX}` indicates the longest relative pathname that could be given if the specified directory is the process' current working directory. A process may not always be able to generate a name that long and use it if a subdirectory in the pathname crosses into a more restrictive file system.

The value returned for the variable `_POSIX_CHOWN_RESTRICTED` also applies to directories that do not have file systems mounted on them. The value may change when crossing a mount point, so applications that need to know should check for each directory. (An even easier check is to try the `chown()` function and look for an error in case it happens.)

Unlike the values returned by `sysconf()`, the pathname-oriented variables are potentially more volatile and are not guaranteed to remain constant throughout the process' lifetime. For
example, in between two calls to `pathconf()`, the file system in question may have been
unmounted and remounted with different characteristics.

Also note that most of the errors are optional. If one of the variables always has the same value
on an implementation, the implementation need not look at `path` or `fildes` to return that value and
is, therefore, not required to detect any of the errors except the meaning of `[EINVAL]` that
indicates that the value of `name` is not valid for that variable.

If the value of any of the limits is unspecified (logically infinite), they will not be defined in
`<limits.h>` and the `pathconf()` and `fpathconf()` functions return −1 without changing `errno`. This
can be distinguished from the case of giving an unrecognized `name` argument because `errno` is set
to `[EINVAL]` in this case.

Since −1 is a valid return value for the `pathconf()` and `fpathconf()` functions, applications should
set `errno` to zero before calling them and check `errno` only if the return value is −1.

For the case of `{SYMLINK_MAX}`, since both `pathconf()` and `open()` follow symbolic links, there
is no way that `path` or `fildes` could refer to a symbolic link.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`confstr()`, `sysconf()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<limits.h>`, `<unistd.h>`,
the Shell and Utilities volume of IEEE Std 1003.1-2001

**CHANGE HISTORY**

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

**Issue 5**

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.
Large File Summit extensions are added.

**Issue 6**

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

- The DESCRIPTION is updated to include `[FILESIZEBITS]`.
- The `[ELOOP]` mandatory error condition is added.
- A second `[ENAMETOOLONG]` is added as an optional error condition.

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

- The `_PC_SYMLINK_MAX` entry is added to the table in the DESCRIPTION.

The following `pathconf()` variables and their associated names are added for alignment with
IEEE Std 1003.1d-1999:

- `[POSIX_ALLOC_SIZE_MIN]`
- `[POSIX_REC_INCR_XFER_SIZE]`
- `[POSIX_REC_MAX_XFER_SIZE]`
- `[POSIX_REC_MIN_XFER_SIZE]`
- `[POSIX_REC_XFER_ALIGN]`
**NAME**
fpclassify — classify real floating type

**SYNOPSIS**
```
#include <math.h>

int fpclassify(real-floating x);
```

**DESCRIPTION**
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 `fpclassify()` macro shall classify its argument value as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.

**RETURN VALUE**
The `fpclassify()` macro shall return the value of the number classification macro appropriate to the value of its argument.

**ERRORS**
No errors are defined.

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`isfinite()`, `isinf()`, `isnan()`, `isnormal()`, `signbit()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<math.h>`

**CHANGE HISTORY**
NAME
fprintf, printf, snprintf, sprintf — print formatted output

SYNOPSIS
#include <stdio.h>

int fprintf(FILE *restrict stream, const char *restrict format, ...);
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n, const char *restrict format, ...);
int sprintf(char *restrict s, const char *restrict format, ...);

DESCRIPTION
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 fprintf() function shall place output on the named output stream. The printf() function shall place output on the standard output stream stdout. The sprintf() function shall place output followed by the null byte, '\0', in consecutive bytes starting at *s; it is the user's responsibility to ensure that enough space is available.

The snprintf() function shall be equivalent to sprintf(), with the addition of the n argument which states the size of the buffer referred to by s. If n is zero, nothing shall be written and s may be a null pointer. Otherwise, output bytes beyond the n-1st shall be discarded instead of being written to the array, and a null byte is written at the end of the bytes actually written into the array.

If copying takes place between objects that overlap as a result of a call to sprintf() or snprintf(), the results are undefined.

Each of these functions converts, formats, and prints its arguments under control of the format.

The format is a character string, beginning and ending in its initial shift state, if any. The format is composed of zero or more directives: ordinary characters, which are simply copied to the output stream, and conversion specifications, each of which shall result 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 shall be evaluated but are otherwise ignored.

Conversions can be applied to the nth 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], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

The format can contain either numbered argument conversion 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 string are undefined. When numbered argument specifications are used, specifying the Nth argument requires that all the leading arguments, from the first to the (N-1)th, are specified in the format string.

In format strings containing the "%n$" form of conversion specification, numbered arguments in the argument list can be referenced from the format string as many times as required.

In format strings containing the % form of conversion specification, each conversion specification uses the first unused argument in the argument list.
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 (`.`).

Each conversion specification is introduced by the `'%` character or by the character sequence `"%n$"`, after which the following appear in sequence:

- Zero or more flags (in any order), which modify the meaning of the conversion specification.
- 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.
- 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.
- An optional length modifier that specifies the size of the argument.
- A conversion specifier character that indicates the type of conversion to be applied.

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$"` form of a conversion specification, a field width or precision may be indicated by the sequence `"*n$"`, 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:

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The flag characters and their meanings are:

- `'`: 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 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
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.

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 's' flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined.

The length modifiers and their meanings are:

- **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.

- **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.

- **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.

- **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.

- **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.

- **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.

- **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.

- **L** Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a long double argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.
The conversion specifiers and their meanings are:

- **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 is 1. The result of converting zero with an explicit precision of zero shall be no characters.

- **o**: The unsigned argument shall be converted to unsigned octal format in the style "dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

- **u**: The unsigned argument shall be converted to unsigned decimal format in the style "dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

- **x**: The unsigned argument shall be converted to unsigned hexadecimal format in the style "dddd"; the letters "abcdef" are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

- **X**: Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead of "abcdef".

- **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. The low-order digit shall be rounded in an implementation-defined manner. The F conversion specifier produces "INF", "INFINITY", or "NAN" instead of "inf", "infinity", or "nan", respectively.

- **e, E**: The double argument shall be converted in the style "[-]d.ddde±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.

- **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 or a ‘#’ flag is present.
A double argument representing an infinity or NaN shall be converted in the style of
an f or F conversion specifier.
A double argument representing a floating-point number shall be converted in the
style "[-]0x.hhhhp±d", where there is one hexadecimal digit (which shall be non-
zero if the argument is a normalized floating-point number and is otherwise
unspecified) before the decimal-point character and the number of hexadecimal digits
after it is 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
character shall appear. The letters "abcdef" shall be 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.
A double argument representing an infinity or NaN shall be converted in the style of
an f or F conversion specifier.

The int argument shall be converted to an unsigned char, and the resulting byte shall
be written.
If an l (ell) qualifier is present, the wint_t argument shall be converted as if by an ls
conversion specification with no precision and an argument that points to a two-
element array of type wchar_t, the first element of which contains the wint_t argument
to the ls conversion specification and the second element contains a null wide
character.
The argument shall be a pointer to an array of char. Bytes from the array shall be
written up to (but not including) any terminating null byte. If the precision is specified,
no more than that many bytes 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 byte.

If an l (ell) qualifier is present, the argument shall be a pointer to an array of type
wchar_t. Wide characters from the array shall be converted to characters (each as if by
a call to the wctomb() function, with the conversion state described by an mbstate_t
object initialized to zero before the first wide character is converted) up to and
including a terminating null wide character. The resulting characters shall be written
up to (but not including) the terminating null character (byte). If no precision is
specified, the application shall ensure that the array contains a null wide character. If a
precision is specified, no more than that many characters (bytes) shall be written
(including shift sequences, if any), and the array shall contain a null wide character if,
to equal the character sequence length given by the precision, the function would need
to access a wide character one past the end of the array. In no case shall a partial
character be written.
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.

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 `fprintf()` functions. No argument is converted.

| XSI C | Equivalent to `lc`. |
| XSI S | Equivalent to `ls`. |

Print a `'%'` character; no argument is converted. The complete conversion specification shall be `%%`.

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.

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 `fprintf()` and `printf()` are printed as if `fputc()` had been called.

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.

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.

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.

The `st_ctime` and `st_mtime` fields of the file shall be marked for update between the call to a successful execution of `fprintf()` or `printf()` and the next successful completion of a call to `fflush()` or `fclose()` on the same stream or a call to `exit()` or `abort()`.

RETURN VALUE

Upon successful completion, the `fprintf()` and `printf()` functions shall return the number of bytes transmitted.

Upon successful completion, the `sprintf()` function shall return the number of bytes written to `s`, excluding the terminating null byte.

Upon successful completion, the `snprintf()` function shall return the number of bytes that would be written to `s` had \( n \) been sufficiently large excluding the terminating null byte.

If an output error was encountered, these functions shall return a negative value.

If the value of \( n \) is zero on a call to `snprintf()`, nothing shall be written, the number of bytes that would have been written had \( n \) been sufficiently large excluding the terminating null shall be returned, and `s` may be a null pointer.
For the conditions under which \texttt{fprintf()} and \texttt{printf()} fail and may fail, refer to \texttt{fputc()} or \texttt{fputwc()}.

In addition, all forms of \texttt{fprintf()} may fail if:

\begin{itemize}
  \item [EILSEQ] A wide-character code that does not correspond to a valid character has been detected.
  \item [EINVAL] There are insufficient arguments.
\end{itemize}

The \texttt{printf()} and \texttt{fprintf()} functions may fail if:

\begin{itemize}
  \item [ENOMEM] Insufficient storage space is available.
\end{itemize}

The \texttt{snprintf()} function shall fail if:

\begin{itemize}
  \item [EOVERFLOW] The value of \texttt{n} is greater than \texttt{INT_MAX} or the number of bytes needed to hold the output excluding the terminating null is greater than \texttt{INT_MAX}.
\end{itemize}

\textbf{EXAMPLES}

\textbf{Printing Language-Independent Date and Time}

The following statement can be used to print date and time using a language-independent format:

\begin{verbatim}
printf(format, weekday, month, day, hour, min);
\end{verbatim}

For American usage, \texttt{format} could be a pointer to the following string:

\begin{verbatim}
"%s, %s %d, %d:%.2d\n"
\end{verbatim}

This example would produce the following message:

\begin{verbatim}
Sunday, July 3, 10:02
\end{verbatim}

For German usage, \texttt{format} could be a pointer to the following string:

\begin{verbatim}
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
\end{verbatim}

This definition of \texttt{format} would produce the following message:

\begin{verbatim}
Sonntag, 3. Juli, 10:02
\end{verbatim}

\textbf{Printing File Information}

The following example prints information about the type, permissions, and number of links of a specific file in a directory.

The first two calls to \texttt{printf()} use data decoded from a previous \texttt{stat()} call. The user-defined \texttt{strperm()} function shall return a string similar to the one at the beginning of the output for the following command:

\begin{verbatim}
ls -l
\end{verbatim}

The next call to \texttt{printf()} outputs the owner's name if it is found using \texttt{getpwuid()}; the \texttt{getpwuid()} function shall return a \texttt{passwd} structure from which the name of the user is extracted. If the user name is not found, the program instead prints out the numeric value of the user ID.

The next call prints out the group name if it is found using \texttt{getgrgid()}; \texttt{getgrgid()} is very similar to \texttt{getpwuid()} except that it shall return group information based on the group number. Once again, if the group is not found, the program prints the numeric value of the group for the entry.
The final call to `printf()` prints the size of the file.

```c
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>

char *strperm (mode_t);
...
struct stat statbuf;
struct passwd *pwd;
struct group *grp;
...
printf("%10.10s", strperm (statbuf.st_mode));
printf("%4d", statbuf.st_nlink);
if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
    printf(" %−8.8s", pwd->pw_name);
else
    printf(" %−8ld", (long) statbuf.st_uid);
if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %−8.8s", grp->gr_name);
else
    printf(" %−8ld", (long) statbuf.st_gid);
printf("%9jd", (intmax_t) statbuf.st_size);
...
```

### Printing a Localized Date String

The following example gets a localized date string. The `nl_langinfo()` function shall return the localized date string, which specifies the order and layout of the date. The `strftime()` function takes this information and, using the `tm` structure for values, places the date and time information into `datestring`. The `printf()` function then outputs `datestring` and the name of the entry.

```c
#include <stdio.h>
#include <time.h>
#include <langinfo.h>
...
 struct dirent *dp;
 struct tm *tm;
 char datestring[256];
...
 strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);
 printf(" %s %s\n", datestring, dp->d_name);
...
Printing Error Information

The following example uses `fprintf()` to write error information to standard error.

In the first group of calls, the program tries to open the password lock file named `LOCKFILE`. If the file already exists, this is an error, as indicated by the `O_EXCL` flag on the `open()` function. If the call fails, the program assumes that someone else is updating the password file, and the program exits.

The next group of calls saves a new password file as the current password file by creating a link between `LOCKFILE` and the new password file `PASSWDFILE`.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
...
int pfd;
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
    {
        fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
        exit(1);
    }
...
if (link(LOCKFILE, PASSWDFILE) == -1) {
    fprintf(stderr, "Link error: %s\n", strerror(errno));
    exit(1);
}
...
```

Printing Usage Information

The following example checks to make sure the program has the necessary arguments, and uses `fprintf()` to print usage information if the expected number of arguments is not present.

```c
#include <stdio.h>
#include <stdlib.h>
...
char *Options = "hdbtl";
...
if (argc < 2) {
    fprintf(stderr, "Usage: %s -%s <file\n", argv[0], Options);
    exit(1);
}
```
Formatting a Decimal String

The following example prints a key and data pair on stdout. Note use of the ‘*’ (asterisk) in the format string; this ensures the correct number of decimal places for the element based on the number of elements requested.

```c
#include <stdio.h>
...
long i;
char *keystr;
int elementlen, len;
...
while (len < elementlen) {
    printf("%s Element%0*ld\n", keystr, elementlen, i);
...
}
```

Creating a Filename

The following example creates a filename using information from a previous getpwnam() function that returned the HOME directory of the user.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
...
char filename[PATH_MAX+1];
struct passwd *pw;
...
sprintf(filename, "%s/%d.out", pw->pw_dir, getpid());
...
```

Reporting an Event

The following example loops until an event has timed out. The pause() function waits forever unless it receives a signal. The fprintf() statement should never occur due to the possible return values of pause().

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
...
while (!event_complete) {
    ...
    if (pause() != -1 || errno != EINTR)
        fprintf(stderr, "pause: unknown error: %s\n", strerror(errno));
}
```
Printing Monetary Information

The following example uses `strfmon()` to convert a number and store it as a formatted monetary string named `convbuf`. If the first number is printed, the program prints the format and the description; otherwise, it just prints the number.

```c
#include <monetary.h>
#include <stdio.h>
...
struct tblfmt {
    char *format;
    char *description;
};

struct tblfmt table[] = {
    { "%n", "default formatting" },
    { "%11n", "right align within an 11 character field" },
    { "%#5n", "aligned columns for values up to 99999" },
    { "%=*#5n", "specify a fill character" },
    { "%==0#5n", "fill characters do not use grouping" },
    { "%=5.0n", "round off to whole units" },
    { "%=5.4n", "increase the precision" },
    { " %^#5n", "use an alternative pos/neg style" },
    { "%!(#5n", "disable the currency symbol" },
};
...
```

```c
float input[3];
int i, j;
char convbuf[100];
...
strfmon(convbuf, sizeof(convbuf), table[i].format, input[j]);
if (j == 0) {
    printf("%s%s%s\n", table[i].format,
            convbuf, table[i].description);
} else {
    printf("%s\n", convbuf);
}
```

Printing Wide Characters

The following example prints a series of wide characters. Suppose that "L @ @" expands to three bytes:

```c
wchar_t wz [3] = L"@@"; // Zero-terminated
wchar_t wn [3] = L"@@@"; // Unterminated
fprintf (stdout, "%ls", wz); // Outputs 6 bytes
fprintf (stdout, "%ls", wn); // Undefined because wn has no terminator
fprintf (stdout, "%4ls", wz); // Outputs 3 bytes
fprintf (stdout, "%4ls", wn); // Outputs 3 bytes; no terminator needed
fprintf (stdout, "%9ls", wz); // Outputs 6 bytes
```
fprintf(stdout, "%9ls", wn); // Outputs 9 bytes; no terminator needed
fprintf(stdout, "%10ls", wz); // Outputs 6 bytes
fprintf(stdout, "%10ls", wn); // Undefined because wn has no terminator

In the last line of the example, after processing three characters, nine bytes have been output.
The fourth character must then be examined to determine whether it converts to one byte or more. If it converts to more than one byte, the output is only nine bytes. Since there is no fourth character in the array, the behavior is undefined.

APPLICATION USAGE
If the application calling `fprintf()` has any objects of type `wint_t` or `wchar_t`, it must also include the `<wchar.h>` header to have these objects defined.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
`fputc()`, `fscanf()`, `setlocale()`, `strfmon()`, `wcrtomb()`, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, `<stdio.h>`, `<wchar.h>`

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the l (ell) qualifier can now be used with c and s conversion specifiers.
The `snprintf()` function is new in Issue 5.

Issue 6
Extensions beyond the ISO C standard are marked.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The following changes are made for alignment with the ISO/IEC 9899:1999 standard:
The prototypes for `fprintf()`, `printf()`, `snprintf()`, and `sprintf()` are updated, and the XSI shading is removed from `snprintf()`.
The description of `snprintf()` is aligned with the ISO C standard. Note that this supersedes the `snprintf()` description in The Open Group Base Resolution bwg98-006, which changed the behavior from Issue 5.
The DESCRIPTION is updated.
The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion specification” consistently.
An example of printing wide characters is added.
fputc()

NAME
fputc — put a byte on a stream

SYNOPSIS
#include <stdio.h>

int fputc(int c, FILE *stream);

DESCRIPTION
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

The \texttt{fputc()} function shall write the byte specified by \texttt{c} (converted to an \texttt{unsigned char}) to the
output stream pointed to by \texttt{stream}, at the position indicated by the associated file-position
indicator for the stream (if defined), and shall advance the indicator appropriately. If the file
cannot support positioning requests, or if the stream was opened with append mode, the byte
shall be appended to the output stream.

The \texttt{st_ctime} and \texttt{st_mtime} fields of the file shall be marked for update between the successful
execution of \texttt{fputc()} and the next successful completion of a call to \texttt{fflush()} or \texttt{fclose()} on the same
stream or a call to \texttt{exit()} or \texttt{abort()}.

RETURN VALUE
Upon successful completion, \texttt{fputc()} shall return the value it has written. Otherwise, it shall
return EOF, the error indicator for the stream shall be set, and \texttt{errno} shall be set to indicate the
error.

ERRORS
The \texttt{fputc()} function shall fail if either the \texttt{stream} is unbuffered or the \texttt{stream}'s buffer needs to be
flushed, and:

- \texttt{[EAGAIN]} The O\_NONBLOCK flag is set for the file descriptor underlying \texttt{stream} and the
  process would be delayed in the write operation.
- \texttt{[EBADF]} The file descriptor underlying \texttt{stream} is not a valid file descriptor open for
  writing.
- \texttt{[EFBIG]} An attempt was made to write to a file that exceeds the maximum file size.
- \texttt{[EFBIG]} An attempt was made to write to a file that exceeds the process' file size limit.
- \texttt{[EFBIG]} The file is a regular file and an attempt was made to write at or beyond the
  offset maximum.
- \texttt{[EINTR]} The write operation was terminated due to the receipt of a signal, and no data
  was transferred.
- \texttt{[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.
- \texttt{[ENOSPC]} There was no free space remaining on the device containing the file.
- \texttt{[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.

The \texttt{fputc()} function may fail if:
fputc()

[ENOMEM] Insufficient storage space is available.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
ferror(), fopen(), getrlimit(), putc(), puts(), setbuf(), ulimit(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
Large File Summit extensions are added.

Issue 6
Extensions beyond the ISO C standard are marked.

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

• The [EIO] and [EFBIG] mandatory error conditions are added.
• The [ENOMEM] and [ENXIO] optional error conditions are added.
**NAME**

`fputs` — put a string on a stream

**SYNOPSIS**

```c
#include <stdio.h>

int fputs(const char *restrict s, FILE *restrict stream);
```

**DESCRIPTION**

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 `fputs()` function shall write the null-terminated string pointed to by `s` to the stream pointed to by `stream`. The terminating null byte shall not be written.

The `st_ctime` and `st_mtime` fields of the file shall be marked for update between the successful execution of `fputs()` and the next successful completion of a call to `fflush()` or `fclose()` on the same stream or a call to `exit()` or `abort()`.

**RETURN VALUE**

Upon successful completion, `fputs()` shall return a non-negative number. Otherwise, it shall return EOF, set an error indicator for the stream, and set `errno` to indicate the error.

**ERRORS**

Refer to `fputc()`.

**EXAMPLES**

**Printing to Standard Output**

The following example gets the current time, converts it to a string using `localtime()` and `asctime()`, and prints it to standard output using `fputs()`. It then prints the number of minutes to an event for which it is waiting.

```c
#include <time.h>
#include <stdio.h>
...
int now;
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
fputs(asctime(localtime(&now)), stdout);
printf("There are still %d minutes to the event.\n", minutes_to_event);
...
```

**APPLICATION USAGE**

The `puts()` function appends a `<newline>` while `fputs()` does not.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.
SEE ALSO
fopen(),putc(),puts(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.

The fputs() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
fputwc — put a wide-character code on a stream

SYNOPSIS
#include <stdio.h>
#include <wchar.h>

wint_t fputwc(wchar_t wc, FILE *stream);

DESCRIPTION
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

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.

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().

RETURN VALUE
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.

ERRORS
The fputwc() function shall fail if either the stream is unbuffered or data in the stream’s buffer
needs to be written, and:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying stream and the
process would be delayed in the write operation.

[EBADF] The file descriptor underlying stream is not a valid file descriptor open for
writing.

[EFBIG] An attempt was made to write to a file that exceeds the maximum file size or
the process’ file size limit.

[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.

[EILSEQ] The wide-character code wc does not correspond to a valid character.

[EINTR] The write operation was terminated due to the receipt of a signal, and no data
was transferred.

[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.

[ENOSPC] There was no free space remaining on the device containing the file.

[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.
The `fputwc()` function may fail if:

- `[ENOMEM]` Insufficient storage space is available.
- `[ENXIO]` A request was made of a nonexistent device, or the request was outside the capabilities of the device.

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`ferror()`, `fopen()`, `setbuf()`, `ulimit()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`, `<wchar.h>`

**CHANGE HISTORY**
First released in Issue 4. Derived from the MSE working draft.

**Issue 5**
Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument `wc` is changed from `wint_t` to `wchar_t`.

The Optional Header (OH) marking is removed from `<stdio.h>`.

Large File Summit extensions are added.

**Issue 6**
Extensions beyond the ISO C standard are marked.

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

- The `[EFBIG]` and `[EIO]` mandatory error conditions are added.
- The `[ENOMEM]` and `[ENXIO]` optional error conditions are added.
fputws()

NAME
fputws — put a wide-character string on a stream

SYNOPSIS
#include <stdio.h>
#include <wchar.h>

int fputws(const wchar_t *restrict ws, FILE *restrict stream);

DESCRIPTION
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

The fputws() function shall write a character string corresponding to the (null-terminated)
wide-character string pointed to by ws to the stream pointed to by stream. No character
corresponding to the terminating null wide-character code shall be written.

The st_ctime and st_mtime fields of the file shall be marked for update between the successful
execution of fputws() and the next successful completion of a call to fflush() or fclose() on the
same stream or a call to exit() or abort().

RETURN VALUE
Upon successful completion, fputws() shall return a non-negative number. Otherwise, it shall
return −1, set an error indicator for the stream, and set errno to indicate the error.

ERRORS
Refer to fputwc().

APPLICATION USAGE
The fputws() function does not append a <newline>.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fopen(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.

Issue 5
The Optional Header (OH) marking is removed from <stdio.h>.

Issue 6
Extensions beyond the ISO C standard are marked.

The fputws() prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.
NAME
fread — binary input

SYNOPSIS
#include <stdio.h>
size_t fread(void *restrict ptr, size_t size, size_t nitems,
        FILE *restrict stream);

DESCRIPTION
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
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.
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(), fgetws(),
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.
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.
Otherwise, if a read error occurs, the error indicator for the stream shall be set, and errno shall be
set to indicate the error.

EXAMPLES
Reading from a Stream
The following example reads a single element from the fp stream into the array pointed to by buf.
#include <stdio.h>
...size_t bytes_read;
char buf[100];
FILE *fp;
...
bytes_read = fread(buf, sizeof(buf), 1, fp);
...

APPLICATION USAGE
The ferror() or feof() functions must be used to distinguish between an error condition and an
end-of-file condition.
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.
or by the same application on a different processor.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
feof(), ferror(), fgetc(), fopen(), gets(), scanf(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

• The fread() prototype is updated.

• The DESCRIPTION is updated to describe how the bytes from a call to fgetc() are stored.
**NAME**
free — free allocated memory

**SYNOPSIS**
#include <stdlib.h>
void free(void *ptr);

**DESCRIPTION**
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 `free()` function shall cause the space pointed to by `ptr` to be deallocated; that is, made available for further allocation. If `ptr` is a null pointer, no action shall occur. Otherwise, if the argument does not match a pointer earlier returned by the `calloc()`, `malloc()`, `posix_memalign()`, `realloc()`, or `strdup()` function, or if the space has been deallocated by a call to `free()` or `realloc()`, the behavior is undefined.

Any use of a pointer that refers to freed space results in undefined behavior.

**RETURN VALUE**
The `free()` function shall not return a value.

**ERRORS**
No errors are defined.

**EXAMPLES**
None.

**APPLICATION USAGE**
There is now no requirement for the implementation to support the inclusion of `<malloc.h>`.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
calloc(), malloc(), realloc(), the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 6**
Reference to the `valloc()` function is removed.
NAME
freeaddrinfo, getaddrinfo — get address information

SYNOPSIS
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo(struct addrinfo *ai);
int getaddrinfo(const char *restrict nodename,
    const char *restrict servname,
    const struct addrinfo *restrict hints,
    struct addrinfo **restrict res);

DESCRIPTION
The freeaddrinfo() function shall free one or more addrinfo structures returned by getaddrinfo(), along with any additional storage associated with those structures. If the ai_next field of the structure is not null, the entire list of structures shall be freed. The freeaddrinfo() function shall support the freeing of arbitrary sublists of an addrinfo list originally returned by getaddrinfo().

The getaddrinfo() function shall translate the name of a service location (for example, a host name) and/or a service name and shall return a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.

The freeaddrinfo() and getaddrinfo() functions shall be thread-safe.

The nodename and servname arguments are either null pointers or pointers to null-terminated strings. One or both of these two arguments shall be supplied by the application as a non-null pointer.

The format of a valid name depends on the address family or families. If a specific family is not given and the name could be interpreted as valid within multiple supported families, the implementation shall attempt to resolve the name in all supported families and, in absence of errors, one or more results shall be returned.

If the nodename argument is not null, it can be a descriptive name or can be an address string. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, valid descriptive names include host names. If the specified address family is AF_INET or AF_UNSPEC, address strings using Internet standard dot notation as specified in inet_addr() are valid.

If the specified address family is AF_INET6 or AF_UNSPEC, standard IPv6 text forms described in inet_ntop() are valid.

If nodename is not null, the requested service location is named by nodename; otherwise, the requested service location is local to the caller.

If servname is null, the call shall return network-level addresses for the specified nodename. If servname is not null, it is a null-terminated character string identifying the requested service. This can be either a descriptive name or a numeric representation suitable for use with the address family or families. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, the service can be specified as a string specifying a decimal port number.

If the hints argument is not null, it refers to a structure containing input values that may direct the operation by providing options and by limiting the returned information to a specific socket type, address family, and/or protocol. In this hints structure every member other than ai_flags, ai_family, ai_socktype, and ai_protocol shall be set to zero or a null pointer. A value of AF_UNSPEC for ai_family means that the caller shall accept any address family. A value of zero for ai_socktype means that the caller shall accept any socket type. A value of zero for ai_protocol means that the caller shall accept any protocol. If hints is a null pointer, the behavior shall be as if
it referred to a structure containing the value zero for the `ai_flags`, `ai_socktype`, and `ai_protocol` fields, and AF_UNSPEC for the `ai_family` field.

The `ai_flags` field to which the `hints` parameter points shall be set to zero or be the bitwise-inclusive OR of one or more of the values AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST, and AI_NUMERICSERV.

If the AI_PASSIVE flag is specified, the returned address information shall be suitable for use in binding a socket for accepting incoming connections for the specified service. In this case, if the `nodename` argument is null, then the IP address portion of the socket address structure shall be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE flag is not specified, the returned address information shall be suitable for a call to `connect()` (for a connection-mode protocol) or for a call to `connect()`, `sendto()`, or `sendmsg()` (for a connectionless protocol). In this case, if the `nodename` argument is null, then the IP address portion of the socket address structure shall be set to the loopback address.

If the AI_CANONNAME flag is specified and the `nodename` argument is not null, the function shall attempt to determine the canonical name corresponding to `nodename` (for example, if `nodename` is an alias or shorthand notation for a complete name).

If the AI_NUMERICHOST flag is specified, then a non-null `nodename` string supplied shall be a numeric host address string. Otherwise, an [EAI_NONAME] error is returned. This flag shall prevent any type of name resolution service (for example, the DNS) from being invoked.

If the AI_NUMERICSERV flag is specified, then a non-null `servname` string supplied shall be a numeric port string. Otherwise, an [EAI_NONAME] error shall be returned. This flag shall prevent any type of name resolution service (for example, NIS+) from being invoked.

If the AI_V4MAPPED flag is specified along with an `ai_family` of AF_INET6, then `getaddrinfo()` shall return IPv4-mapped IPv6 addresses on finding no matching IPv6 addresses (`ai_addrlen` shall be 16). The AI_V4MAPPED flag shall be ignored unless `ai_family` equals AF_INET6. If the AI_ALL flag is used with the AI_V4MAPPED flag, then `getaddrinfo()` shall return all matching IPv6 and IPv4 addresses. The AI_ALL flag without the AI_V4MAPPED flag is ignored.

The `ai_socktype` field to which argument `hints` points specifies the socket type for the service, as defined in `socket()`. If a specific socket type is not given (for example, a value of zero) and the service name could be interpreted as valid with multiple supported socket types, the implementation shall attempt to resolve the service name for all supported socket types and, in the absence of errors, all possible results shall be returned. A non-zero socket type value shall limit the returned information to values with the specified socket type.

If the `ai_family` field to which `hints` points has the value AF_UNSPEC, addresses shall be returned for use with any address family that can be used with the specified `nodename` and/or `servname`. Otherwise, addresses shall be returned for use only with the specified address family.

If `ai_family` is not AF_UNSPEC and `ai_protocol` is not zero, then addresses are returned for use only with the specified address family and protocol; the value of `ai_protocol` shall be interpreted as in a call to the `socket()` function with the corresponding values of `ai_family` and `ai_protocol`.

**RETURN VALUE**

A zero return value for `getaddrinfo()` indicates successful completion; a non-zero return value indicates failure. The possible values for the failures are listed in the ERRORS section.

Upon successful return of `getaddrinfo()`, the location to which `res` points shall refer to a linked list of `addrinfo` structures, each of which shall specify a socket address and information for use in creating a socket with which to use that socket address. The list shall include at least one `addrinfo` structure. The `ai_next` field of each structure contains a pointer to the next structure on the list, or a null pointer if it is the last structure on the list. Each structure on the list shall
include values for use with a call to the `socket()` function, and a socket address for use with the `connect()` function or, if the AI_PASSIVE flag was specified, for use with the `bind()` function. The fields `ai_family`, `ai_socktype`, and `ai_protocol` shall be usable as the arguments to the `socket()` function to create a socket suitable for use with the returned address. The fields `ai_addr` and `ai_addrlen` are usable as the arguments to the `connect()` or `bind()` functions with such a socket, according to the AI_PASSIVE flag.

If `nodename` is not null, and if requested by the AI_CANONNAME flag, the `ai_canonname` field of the first returned `addrinfo` structure shall point to a null-terminated string containing the canonical name corresponding to the input `nodename`; if the canonical name is not available, then `ai_canonname` shall refer to the `nodename` argument or a string with the same contents. The contents of the `ai_flags` field of the returned structures are undefined.

All fields in socket address structures returned by `getaddrinfo()` that are not filled in through an explicit argument (for example, `sin6_flowinfo`) shall be set to zero.

**Note:** This makes it easier to compare socket address structures.

**ERRORS**

The `getaddrinfo()` function shall fail and return the corresponding value if:

- [EAI_AGAIN] The name could not be resolved at this time. Future attempts may succeed.
- [EAI_BADFLAGS] The `flags` parameter had an invalid value.
- [EAI_FAIL] A non-recoverable error occurred when attempting to resolve the name.
- [EAI_FAMILY] The address family was not recognized.
- [EAI_MEMORY] There was a memory allocation failure when trying to allocate storage for the return value.
- [EAI_NONAME] The name does not resolve for the supplied parameters.
- [EAI_SERVICE] The service passed was not recognized for the specified socket type.
- [EAI_SOCKTYPE] The intended socket type was not recognized.
- [EAI_SYSTEM] A system error occurred; the error code can be found in `errno`.
- [EAI_OVERFLOW] An argument buffer overflowed.

**EXAMPLES**

None.

**APPLICATION USAGE**

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.

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.

**RATIONALE**

None.
FUTURE DIRECTIONS

None.

SEE ALSO
connect(), gai_strerror(), gethostbyname(), getnameinfo(), getservbyname(), socket(), the Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>, <sys/socket.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
The restrict keyword is added to the getaddrinfo() prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
freopen — open a stream

SYNOPSIS
#include <stdio.h>

FILE *freopen(const char *restrict filename, const char *restrict mode,
               FILE *restrict stream);

DESCRIPTION
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
The freopen() function shall first attempt to flush the stream and close any file descriptor
associated with stream. Failure to flush or close the file descriptor successfully shall be ignored.
The error and end-of-file indicators for the stream shall be cleared.
The freopen() function shall open the file whose pathname is the string pointed to by filename and
associate the stream pointed to by stream with it. The mode argument shall be used just as in
fopen().
The original stream shall be closed regardless of whether the subsequent open succeeds.
If filename is a null pointer, the freopen() function shall attempt to change the mode of the stream
to that specified by mode, as if the name of the file currently associated with the stream had been
used. It is implementation-defined which changes of mode are permitted (if any), and under
what circumstances.
After a successful call to the freopen() function, the orientation of the stream shall be cleared, the
encoding rule shall be cleared, and the associated mbstate_t object shall be set to describe an
initial conversion state.
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.

RETURN VALUE
Upon successful completion, freopen() shall return the value of stream. Otherwise, a null pointer
shall be returned, and errno shall be set to indicate the error.

ERRORS
The freopen() function shall fail if:

[EACCES] Search permission is denied on a component of the path prefix, or the file
exists and the permissions specified by mode are denied, or the file does not
exist and write permission is denied for the parent directory of the file to be
created.

[EINTR] A signal was caught during freopen().

[EISDIR] The named file is a directory and mode requires write access.

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

[EMFILE] [OPEN_MAX] file descriptors are currently open in the calling process.

[ENAMETOOLONG] The length of the filename argument exceeds [PATH_MAX] or a pathname
component is longer than [NAME_MAX].
The `freopen()` function may fail if:

- **[EINVAL]** The value of the `mode` argument is not valid.
- **[ELOOP]** More than `{SYMLOOP_MAX}` symbolic links were encountered during resolution of the `path` argument.
- **[ENAMETOOLONG]** Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `{PATH_MAX}`.
- **[ENOMEM]** Insufficient storage space is available.
- **[ENXIO]** A request was made of a nonexistent device, or the request was outside the capabilities of the device.
- **[ETXTBSY]** The file is a pure procedure (shared text) file that is being executed and `mode` requires write access.

### EXAMPLES

#### Directing Standard Output to a File

The following example logs all standard output to the `/tmp/logfile` file.

```c
#include <stdio.h>
...
FILE *fp;
...
fp = freopen ("/tmp/logfile", "a+", stdout);
... 
```

### APPLICATION USAGE

The `freopen()` function is typically used to attach the preopened streams associated with `stdin`, `stdout`, and `stderr` to other files.

### RATIONALE

None.
**FUTURE DIRECTIONS**
None.

**SEE ALSO**
fclose(), fopen(), fdopen(), mbsinit(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**
The DESCRIPTION is updated to indicate that the orientation of the stream is cleared and the conversion state of the stream is set to an initial conversion state by a successful call to the freopen() function.

Large File Summit extensions are added.

**Issue 6**
Extensions beyond the ISO C standard are marked.

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

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support 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.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The freopen() prototype is updated.
- The DESCRIPTION is updated.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

The DESCRIPTION is updated regarding failure to close, changing the "file" to "file descriptor".
# frexp( )

**NAME**
frexp, frexpf, frexpl — extract mantissa and exponent from a double precision number

**SYNOPSIS**
```c
#include <math.h>

double frexp(double num, int *exp);
float frexpf(float num, int *exp);
long double frexpl(long double num, int *exp);
```

**DESCRIPTION**
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.

These functions shall break a floating-point number `num` into a normalized fraction and an integral power of 2. The integer exponent shall be stored in the `int` object pointed to by `exp`.

**RETURN VALUE**
For finite arguments, these functions shall return the value `x`, such that `x` has a magnitude in the interval `1⁄2,1) or 0, and `num` equals `x` times 2 raised to the power `*exp`.

- If `num` is NaN, a NaN shall be returned, and the value of `*exp` is unspecified.
- If `num` is ±0, ±0 shall be returned, and the value of `*exp` shall be 0.
- If `num` is ±Inf, `num` shall be returned, and the value of `*exp` is unspecified.

**ERRORS**
No errors are defined.

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`isnan()`, `ldexp()`, `modf()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<math.h>`

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

**Issue 6**
The `frexpf()` and `frexpl()` functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

NAME

fscanf, scanf, sscanf — convert formatted input

SYNOPSIS

#include <stdio.h>

int fscanf(FILE *restrict stream, const char *restrict format, ...);
int scanf(const char *restrict format, ...);
int sscanf(const char *restrict s, const char *restrict format, ...);

DESCRIPTION

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 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.

Conversions can be applied to the n<sup>th</sup> 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 "%$n"<sup>12</sup>, where n is a decimal integer in the range [1,NL_ARGMAX]. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages. In format strings containing the "%$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.

The format can contain either form of a conversion specification—that is, % or "%$n"<sup>12</sup>—but the two forms cannot be mixed within a single format string. The only exception to this is that % or %* can be mixed with the "%$n" form. When numbered argument specifications are used, specifying the N<sup>th</sup> argument requires that all the leading arguments, from the first to the (N−1)<sup>th</sup>, are pointers.

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 ('.').

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"<sup>12</sup>, after which the following appear in sequence:

- An optional assignment-suppressing character ' *'.
- An optional non-zero decimal integer that specifies the maximum field width.
- An option length modifier that specifies the size of the receiving object.
- A conversion specifier character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.
The `fscanf()` functions shall execute each directive of the format in turn. If a directive fails, as detailed below, the function shall return. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space characters shall be executed by reading input until no more valid input can be read, or up to the first byte which is not a white-space character, which remains unread.

A directive that is an ordinary character shall be executed as follows: the next byte shall be read from the input and compared with the byte that comprises the directive; if the comparison shows that they are not equivalent, the directive shall fail, and the differing and subsequent bytes shall remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive shall fail.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion character. A conversion specification shall be executed in the following steps.

Input white-space characters (as specified by `isspace()`) shall be skipped, unless the conversion specification includes a `, c, C, or n conversion specifier.

An item shall be read from the input, unless the conversion specification includes an n conversion specifier. An input item shall be defined as the longest sequence of input bytes (up to any specified maximum field width, which may be measured in characters or bytes dependent on the conversion specifier) which is an initial subsequence of a matching sequence. The first byte, if any, after the input item shall remain unread. If the length of the input item is 0, the execution of the conversion specification shall fail; this condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a % conversion specifier, the input item (or, in the case of a %n conversion specification, the count of input bytes) shall be converted to a type appropriate to the conversion character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a 's', the result of the conversion shall be placed in the object pointed to by the first argument following the format argument that has not already received a conversion result if the conversion specification is introduced by %, or in the nth argument if introduced by the 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.

The length modifiers and their meanings are:

- **hh** Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to `signed char` or `unsigned char`.
- **h** Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to `short` or `unsigned short`.
- **l** (ell) Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to `long` or `unsigned long`; that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to `double`; or that a following c, s, or [ conversion specifier applies to an argument with type pointer to `wchar_t`.
- **ll** (ell-ell) Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to `long long` or `unsigned long long`. 
Specifies that a following d, i, o, u, x, or n conversion specifier applies to an argument with type pointer to \texttt{inmax_t} or \texttt{uintmax_t}.

Specifies that a following d, i, o, u, x, or n conversion specifier applies to an argument with type pointer to \texttt{size_t} or the corresponding signed integer type.

Specifies that a following d, i, o, u, x, or n conversion specifier applies to an argument with type pointer to \texttt{ptrdiff_t} or the corresponding unsigned type.

Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to \texttt{long double}.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The following conversion specifiers are valid:

\begin{itemize}
\item \texttt{d} Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of \texttt{strtol()} with the value 10 for the \texttt{base} argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to \texttt{int}.
\item \texttt{i} Matches an optionally signed integer, whose format is the same as expected for the subject sequence of \texttt{strtol()} with 0 for the \texttt{base} argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to \texttt{int}.
\item \texttt{o} Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of \texttt{strtoul()} with the value 8 for the \texttt{base} argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to \texttt{unsigned}.
\item \texttt{u} Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of \texttt{strtoul()} with the value 10 for the \texttt{base} argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to \texttt{unsigned}.
\item \texttt{x} Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of \texttt{strtoul()} with the value 16 for the \texttt{base} argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to \texttt{unsigned}.
\item \texttt{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 \texttt{strtod()} in the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to \texttt{float}.
\item \texttt{s} Matches a sequence of bytes that are not white-space characters. The application shall ensure that the corresponding argument is a pointer to the initial byte of an array of \texttt{char}, \texttt{signed char}, or \texttt{unsigned char} large enough to accept the sequence and a terminating null character code, which shall be added automatically.
\end{itemize}

If an \texttt{1 (ell)} qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character shall be converted to a wide character as if by a call to...
the `mbtowc()` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first character is converted. The application shall ensure that the corresponding argument is a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

Matches a non-empty sequence of bytes from a set of expected bytes (the `scanset`). The normal skip over white-space characters shall be suppressed in this case. The application shall ensure that the corresponding argument is a pointer to the initial byte of an array of `char`, `signed char`, or `unsigned char` large enough to accept the sequence and a terminating null byte, which shall be added automatically.

If an `l` (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character in the sequence shall be converted to a wide character as if by a call to the `mbtowc()` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first character is converted. The application shall ensure that the corresponding argument is a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

The conversion specification includes all subsequent bytes in the `format` string up to and including the matching right square bracket (`' ]'`). The bytes between the square brackets (the `scanlist`) comprise the scanset, unless the byte after the left square bracket is a circumflex (`'ˆ'`), in which case the scanset contains all bytes that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with `"[ ]"` or `"[ ]"`, the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise, the first right square bracket is the one that ends the conversion specification. If a `'ˆ'` is in the scanlist and is not the first character, nor the second where the first character is a `'ˆ'`, nor the last character, the behavior is implementation-defined.

Matches a sequence of bytes of the number specified by the field width (1 if no field width is present in the conversion specification). The application shall ensure that the corresponding argument is a pointer to the initial byte of an array of `char`, `signed char`, or `unsigned char` large enough to accept the sequence. No null byte is added. The normal skip over white-space characters shall be suppressed in this case.

If an `l` (ell) qualifier is present, the input shall be a sequence of characters that begins in the initial shift state. Each character in the sequence is converted to a wide character as if by a call to the `mbtowc()` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first character is converted. The application shall ensure that the corresponding argument is a pointer to an array of `wchar_t` large enough to accept the resulting sequence of wide characters. No null wide character is added.

Matches an implementation-defined set of sequences, which shall be the same as the set of sequences that is produced by the `%P` conversion specification of the corresponding `fprintf()` functions. The application shall ensure that the corresponding argument is a pointer to a pointer to `void`. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise, the behavior of the `%P` conversion specification is undefined.

No input is consumed. The application shall ensure that the corresponding argument is a pointer to the integer into which shall be written the number of bytes read from the
input so far by this call to the fscanf() 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 character or a field
 width, the behavior is undefined.

%f
Equivalent to lc.

% Equivalent to ls.

% Matches a single ‘’ character; no conversion or assignment occurs. The complete
 conversion specification shall be %.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers A, E, F, G, and X are also valid and shall be equivalent to a, e, f, g, and
 x, respectively.

If end-of-file is encountered during input, conversion shall be terminated. If end-of-file occurs
 before any bytes matching the current conversion specification (except for %n) have been read
 (other than leading white-space characters, 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.

Reaching the end of the string in sscanf() shall be equivalent to encountering end-of-file for
 fscanf().

If conversion terminates on a conflicting input, the offending input is left unread in the input.

Any trailing white space (including <newline>s) 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.

The fscanf() and scanf() functions 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(), fread(), getc(), getchar(), gets(), fscanf(), or sscanf() using stream that returns data
 not supplied by a prior call to ungetc().

RETURN VALUE

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 errno shall be set
 to indicate the error.

ERRORS

For the conditions under which the fscanf() functions fail and may fail, refer to fgetc() or
 fgetwc().

In addition, fscanf() may fail if:

[EILSEQ] Input byte sequence does not form a valid character.

[EINVAL] There are insufficient arguments.
EXAMPLES

The call:

```c
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 Hamster
```

assigns to \( n \) the value 3, to \( i \) the value 25, to \( x \) the value 5.432, and \( \text{name} \) contains the string "Hamster".

The call:

```c
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

assigns 56 to \( i \), 789.0 to \( x \), skips 0123, and places the string "56\0" in \( \text{name} \). The next call to `getchar()` shall return the character 'a'.

Reading Data into an Array

The following call uses `fscanf()` to read three floating-point numbers from standard input into the input array.

```c
float input[3]; fscanf(stdin, "%f %f %f", input, input+1, input+2);
```

APPLICATION USAGE

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.

RATIONALE

This function is aligned with the ISO/IEC 9899: 1999 standard, and in doing so a few "obvious" things were not included. Specifically, the set of characters allowed in a scanset is limited to single-byte characters. In other similar places, multi-byte characters have been permitted, but for alignment with the ISO/IEC 9899:1999 standard, it has not been done here. Applications needing this could use the corresponding wide-character functions to achieve the desired results.

FUTURE DIRECTIONS

None.

SEE ALSO

`getc()`, `printf()`, `setlocale()`, `strtol()`, `strtol()`, `strtol()`, `wctomb()`, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, `<langinfo.h>`, `<stdio.h>`, `<wchar.h>`

CHANGE HISTORY

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

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the \( l \) (ell) qualifier is now defined for the \( c \), \( s \), and \( l \) conversion specifiers.

The DESCRIPTION is updated to indicate that if infinity and NaN can be generated by the `fprintf()` family of functions, then they are recognized by the `fscanf()` family.
The Open Group Corrigenda U021/7 and U028/10 are applied. These correct several occurrences of ‘‘characters’’ in the text which have been replaced with the term ‘‘bytes’’.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

• The prototypes for fscanf(), scanf(), and sscanf() are updated.
• The DESCRIPTION is updated.
• The hh, ll, j, t, and z length modifiers are added.
• The a, A, and f conversion characters are added.

The DESCRIPTION is updated to use the terms ‘‘conversion specifier’’ and ‘‘conversion specification’’ consistently.
**NAME**
fseek, fseeko — reposition a file-position indicator in a stream

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

int fseek(FILE *stream, long offset, int whence);
CX int fseeko(FILE *stream, off_t offset, int whence);
```

**DESCRIPTION**
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 `fseek()` function shall set the file-position indicator for the stream pointed to by `stream`. If a read or write error occurs, the error indicator for the stream shall be set and `fseek()` fails.

The new position, measured in bytes from the beginning of the file, shall be obtained by adding `offset` to the position specified by `whence`. The specified point is the beginning of the file for `SEEK_SET`, the current value of the file-position indicator for `SEEK_CUR`, or end-of-file for `SEEK_END`.

If the stream is to be used with wide-character input/output functions, the application shall ensure that `offset` is either 0 or a value returned by an earlier call to `ftell()` on the same stream and `whence` is `SEEK_SET`.

A successful call to `fseek()` shall clear the end-of-file indicator for the stream and undo any effects of `ungetc()` and `ungetwc()` on the same stream. After an `fseek()` call, the next operation on an update stream may be either input or output.

If the most recent operation, other than `ftell()`, on a given stream is `fflush()`, the file offset in the underlying open file description shall be adjusted to reflect the location specified by `fseek()`.

The `fseek()` function shall allow the file-position indicator to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads of data in the gap shall return bytes with the value 0 until data is actually written into the gap.

The behavior of `fseek()` on devices which are incapable of seeking is implementation-defined.

The value of the file offset associated with such a device is undefined.

If the stream is writable and buffered data had not been written to the underlying file, `fseek()` shall cause the unwritten data to be written to the file and shall mark the `st_ctime` and `st_mtime` fields of the file for update.

In a locale with state-dependent encoding, whether `fseek()` restores the stream’s shift state is implementation-defined.

The `fseeko()` function shall be equivalent to the `fseek()` function except that the `offset` argument is of type `off_t`.

**RETURN VALUE**

CX The `fseek()` and `fseeko()` functions shall return 0 if they succeed.

CX Otherwise, they shall return −1 and set `errno` to indicate the error.

**ERRORS**

CX The `fseek()` and `fseeko()` functions shall fail if, either the `stream` is unbuffered or the `stream`’s buffer needed to be flushed, and the call to `fseek()` or `fseeko()` causes an underlying `lseek()` or `write()` to be invoked, and:
fseek()

The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.

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.

An attempt was made to write a file that exceeds the maximum file size.

An attempt was made to write a file that exceeds the process' file size limit.

The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

The write operation was terminated due to the receipt of a signal, and no data was transferred.

The whence argument is invalid. The resulting file-position indicator would be set to a negative value.

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.

There was no free space remaining on the device containing the file.

A request was made of a nonexistent device, or the request was outside the capabilities of the device.

For fseek(), the resulting file offset would be a value which cannot be represented correctly in an object of type long.

For fseeko(), the resulting file offset would be a value which cannot be represented correctly in an object of type off_t.

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.

The file descriptor underlying stream is associated with a pipe or FIFO.

None.

None.

None.

None.

None.

None.

None.

fopen(), fsetpos(), ftell(), getrlimit(), lseek(), rewind(), ulimit(), ungetc(), write()}

First released in Issue 1. Derived from Issue 1 of the SVID.
**fseek()**

**System Interfaces**

<table>
<thead>
<tr>
<th>Line</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>14528</td>
<td><strong>Issue 5</strong></td>
</tr>
<tr>
<td>14529</td>
<td>Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.</td>
</tr>
<tr>
<td>14530</td>
<td>Large File Summit extensions are added.</td>
</tr>
<tr>
<td>14532</td>
<td><strong>Issue 6</strong></td>
</tr>
<tr>
<td>14533</td>
<td>Extensions beyond the ISO C standard are marked.</td>
</tr>
<tr>
<td>14534</td>
<td>The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:</td>
</tr>
<tr>
<td>14535</td>
<td>• The fseeko() function is added.</td>
</tr>
<tr>
<td>14536</td>
<td>• The [EFBIG], [EOVERFLOW], and [ENXIO] mandatory error conditions are added.</td>
</tr>
<tr>
<td>14538</td>
<td>The following change is incorporated for alignment with the FIPS requirements:</td>
</tr>
<tr>
<td>14539</td>
<td>• The [EINTR] error is no longer an indication that the implementation does not report partial transfers.</td>
</tr>
<tr>
<td>14541</td>
<td>The DESCRIPTION is updated to avoid use of the term “must” for application requirements.</td>
</tr>
<tr>
<td>14542</td>
<td>The DESCRIPTION is updated to explicitly state that fseek() sets the file-position indicator, and then on error the error indicator is set and fseek() fails. This is for alignment with the ISO/IEC 9899:1999 standard.</td>
</tr>
</tbody>
</table>
NAME
fsetpos — set current file position

SYNOPSIS
#include <stdio.h>

int fsetpos(FILE *stream, const fpos_t *pos);

DESCRIPTION
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 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.

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.

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.

RETURN VALUE
The fsetpos() function shall return 0 if it succeeds; otherwise, it shall return a non-zero value and set errno to indicate the error.

ERRORS
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:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.

[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.

[EFBIG] An attempt was made to write a file that exceeds the maximum file size.

[EFBIG] An attempt was made to write a file that exceeds the process’ file size limit.

[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.

[EINTR] The write operation was terminated due to the receipt of a signal, and no data was transferred.

[EINVAL] The whence argument is invalid. The resulting file-position indicator would be set to a negative value.

[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.

[ENOSPC] There was no free space remaining on the device containing the file.
fsetpos()

System Interfaces

A request was made of a nonexistent device, or the request was outside the
capabilities of the device.

The file descriptor underlying stream is associated with a pipe or FIFO.

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.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fopen(), ftell(), lseek(), rewind(), ungetc(), write(), the Base Definitions volume of
IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 4. Derived from the ISO C standard.

Issue 6
Extensions beyond the ISO C standard are marked.
An additional [ESPIPE] error condition is added for sockets.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The DESCRIPTION is updated to clarify that the error indicator is set for the stream on a read or
write error. This is for alignment with the ISO/IEC 9899: 1999 standard.
NAME
fstat — get file status

SYNOPSIS
#include <sys/stat.h>
int fstat(int fildes, struct stat *buf);

DESCRIPTION
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.

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_mode fields, and only the
S_IRUSR, S_IWUSR, S_IROTH, and S_IWOTH file permission bits need be
valid. The implementation may update other fields and flags.

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_mode fields, and only the
S_IRUSR, S_IWUSR, S_IROTH, and S_IWOTH file permission bits need be
valid. The implementation may update other fields and flags.

The buf argument is a pointer to a stat structure, as defined in <sys/stat.h>, into which
information is placed concerning the file.

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.

An implementation that provides additional or alternative file access control mechanisms may,
under implementation-defined conditions, cause fstat() to fail.

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.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to
indicate the error.

ERRORS
The fstat() function shall fail if:

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

[EIO] An I/O error occurred while reading from the file system.

[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.

The fstat() function may fail if:

[EOVERFLOW] One of the values is too large to store into the structure pointed to by the buf
argument.
EXAMPLES

Obtaining File Status Information

The following example shows how to obtain file status information for a file named `/home/cnd/mod1`. The structure variable `buffer` is defined for the `stat` structure. The `/home/cnd/mod1` file is opened with read/write privileges and is passed to the open file descriptor `fildes`.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct stat buffer;
int status;
...
fildes = open("/home/cnd/mod1", O_RDWR);
status = fstat(fildes, &buffer);
```

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
`lstat()`, `stat()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/stat.h>`, `<sys/types.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.
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.
- The [EIO] mandatory error condition is added.
- The [EOVERFLOW] mandatory error condition is added. This change is to support large files.
- The [EOVERFLOW] optional error condition is added.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that shared memory object semantics apply to typed memory objects.
NAME
fstatvfs, statvfs — get file system information

SYNOPSIS
XSI
#include <sys/statvfs.h>

int fstatvfs(int fildes, struct statvfs *buf);
int statvfs(const char *restrict path, struct statvfs *restrict buf);

DESCRIPTION
The fstatvfs() function shall obtain information about the file system containing the file
referenced by fildes.
The statvfs() function shall obtain information about the file system containing the file named by
path.
For both functions, the buf argument is a pointer to a statvfs structure that shall be filled. Read,
write, or execute permission of the named file is not required.
The following flags can be returned in the f_flag member:
ST_RDONLY Read-only file system.
ST_NOSUID Setuid/setgid bits ignored by exec.
It is unspecified whether all members of the statvfs structure have meaningful values on all file
systems.

RETURN VALUE
Upon successful completion, statvfs() shall return 0. Otherwise, it shall return -1 and set errno to
indicate the error.

ERRORS
The fstatvfs() and statvfs() functions shall fail if:
[EIO] An I/O error occurred while reading the file system.
[EINTR] A signal was caught during execution of the function.
[EOVERFLOW] One of the values to be returned cannot be represented correctly in the
structure pointed to by buf.
The fstatvfs() function shall fail if:
[EBADF] The fildes argument is not an open file descriptor.
The statvfs() function shall fail if:
[EACCES] Search permission is denied on a component of the path prefix.
[ELOOP] A loop exists in symbolic links encountered during resolution of the path
argument.
[ENAMETOOLONG] The length of a pathname exceeds PATH_MAX or a pathname component is
longer than NAME_MAX.
[ENOENT] A component of path does not name an existing file or path is an empty string.
[ENOTDIR] A component of the path prefix of path is not a directory.
The statvfs() function may fail if:
fstatvfs( ) More than [SYMLOOP_MAX] symbolic links were encountered during resolution of the path argument.

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds [PATH_MAX].

EXAMPLES

Obtaining File System Information Using fstatvfs()

The following example shows how to obtain file system information for the file system upon which the file named /home/cnd/mod1 resides, using the fstatvfs() function. The /home/cnd/mod1 file is opened with read/write privileges and the open file descriptor is passed to the fstatvfs() function.

```c
#include <statvfs.h>
#include <fcntl.h>

struct statvfs buffer;
int status;
...
fd = open("/home/cnd/mod1", O_RDWR);
status = fstatvfs(fd, &buffer);
```

Obtaining File System Information Using statvfs()

The following example shows how to obtain file system information for the file system upon which the file named /home/cnd/mod1 resides, using the statvfs() function.

```c
#include <statvfs.h>

struct statvfs buffer;
int status;
...
status = statvfs("/home/cnd/mod1", &buffer);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod(), chown(), creat(), dup(), exec, fcntl(), link(), mknod(), open(), pipe(), read(), time(), unlink(), utime(), write(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/statvfs.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Large File Summit extensions are added.
Issue 6

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

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

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.
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` in an implementation-defined manner. The `fsync()` function shall not return until the system has completed that action or until an error is detected.

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.

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.

The `fsync()` function shall fail if:

- `[EBADF]` The `fildes` argument is not a valid descriptor.
- `[EINTR]` The `fsync()` function was interrupted by a signal.
- `[EINVAL]` The `fildes` argument does not refer to a file on which this operation is possible.
- `[EIO]` An I/O error occurred while reading from or writing to the file system.

In the event that any of the queued I/O operations fail, `fsync()` shall return the error conditions defined for `read()` and `write()`.

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.

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.

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
conformance document should identify at least that one configuration exists (and how to obtain that configuration) where this can be assured for at least some files that the user can select to use for critical data. It is not intended that an exhaustive list is required, but rather sufficient information is provided so that if critical data needs to be saved, the user can determine how the system is to be configured to allow the data to be written to non-volatile storage.

It is reasonable to assert that the key aspects of \texttt{fsync()} are unreasonable to test in a test suite. That does not make the function any less valuable, just more difficult to test. A formal conformance test should probably force a system crash (power shutdown) during the test for this condition, but it needs to be done in such a way that automated testing does not require this to be done except when a formal record of the results is being made. It would also not be unreasonable to omit testing for \texttt{fsync()}, allowing it to be treated as a quality-of-implementation issue.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

\texttt{sync()}, the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<unistd.h>}

**CHANGE HISTORY**

First released in Issue 3.

**Issue 5**

Aligned with \texttt{fsync()} in the POSIX Realtime Extension. Specifically, the DESCRIPTION and RETURN VALUE sections are much expanded, and the ERRORS section is updated to indicate that \texttt{fsync()} can return the error conditions defined for \texttt{read()} and \texttt{write()}.

**Issue 6**

This function is marked as part of the File Synchronization option.

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

- The [EINVAL] and [EIO] mandatory error conditions are added.
NAME
ftell, ftello — return a file offset in a stream

SYNOPSIS
#include <stdio.h>

long ftell(FILE *stream);

off_t ftello(FILE *stream);

DESCRIPTION
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 ftell() function shall obtain the current value of the file-position indicator for the stream pointed to by stream.

The ftello() function shall be equivalent to ftell(), except that the return value is of type off_t.

RETURN VALUE
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.

Otherwise, ftell() and ftello() shall return -1, cast to long and off_t respectively, and set errno to indicate the error.

ERRORS
The ftell() and ftello() functions shall fail if:

[EBADF] The file descriptor underlying stream is not an open file descriptor.

[EOVERFLOW] For ftell(), the current file offset cannot be represented correctly in an object of type long.

[EOVERFLOW] For ftello(), the current file offset cannot be represented correctly in an object of type off_t.

[ESPIPE] The file descriptor underlying stream is associated with a pipe or FIFO.

The ftell() function may fail if:

[ESPIPE] The file descriptor underlying stream is associated with a socket.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fgetpos(), fopen(), fseek(), lseek(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>
Change History

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

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are marked.

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

- The ftello() function is added.
- The [EOVERFLOW] error conditions are added.
- An additional [ESPIPE] error condition is added for sockets.
NAME
ftime — get date and time (LEGACY)

SYNOPSIS
#include <sys/timeb.h>

int ftime(struct timeb *tp);

DESCRIPTION
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.
The system clock need not have millisecond granularity. Depending on any granularity
(particularly a granularity of one) renders code non-portable.

RETURN VALUE
Upon successful completion, the ftime() function shall return 0; otherwise, −1 shall be returned.

ERRORS
No errors are defined.

EXAMPLES
Getting the Current Time and Date
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.
#include <sys/timeb.h>
struct timeb tp;
int status;
... status = ftime(&tp);

APPLICATION USAGE
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().

RATIONALE
None.

FUTURE DIRECTIONS
This function may be withdrawn in a future version.

SEE ALSO
clock_getres(), ctime(), gettimeofday(), time(), the Base Definitions volume of
IEEE Std 1003.1-2001, <sys/timeb.h>

CHANGE HISTORY
First released in Issue 4, Version 2.
Moved from X/OPEN UNIX extension to BASE.

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

This function is marked LEGACY.

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.
NAME
ftok — generate an IPC key

SYNOPSIS
#include <sys/ipc.h>

key_t ftok(const char *path, int id);

DESCRIPTION
The ftok() function shall return a key based on path and id that is usable in subsequent calls to
msgget(), semget(), and shmget(). The application shall ensure that the path argument is the
pathname of an existing file that the process is able to stat().

The ftok() function shall return the same key value for all paths that name the same file, when
called with the same id value, and return different key values when called with different id
values or with paths that name different files existing on the same file system at the same time. It
is unspecified whether ftok() shall return the same key value when called again after the file
named by path is removed and recreated with the same name.

Only the low-order 8-bits of id are significant. The behavior of ftok() is unspecified if these bits
are 0.

RETURN VALUE
Upon successful completion, ftok() shall return a key. Otherwise, ftok() shall return (key_t)−1
and set errno to indicate the error.

ERRORS
The ftok() function shall fail if:

[EACCES] Search permission is denied for a component of the path prefix.

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

[ENAMETOOLONG] The length of the path argument exceeds {PATH_MAX} or a pathname
component is longer than {NAME_MAX},

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

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

The ftok() function may fail if:

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

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
whose length exceeds {PATH_MAX}. 
### Examples

#### Getting an IPC Key

The following example gets a unique key that can be used by the IPC functions `semget()`, `msgget()`, and `shmget()`. The key returned by `ftok()` for this example is based on the ID value `S` and the pathname `/tmp`.

```c
#include <sys/ipc.h>
...  
key_t key;
char *path = "/tmp";
int id = 'S';
key = ftok(path, id);
```

#### Saving an IPC Key

The following example gets a unique key based on the pathname `/tmp` and the ID value `a`. It also assigns the value of the resulting key to the `semkey` variable so that it will be available to a later call to `semget()`, `msgget()`, or `shmget()`.

```c
#include <sys/ipc.h>
...  
key_t semkey;
if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {  
    perror("IPC error: ftok"); exit(1);
}
```

### Application Usage

For maximum portability, `id` should be a single-byte character.

### Rationale

None.

### Future Directions

None.

### See Also

`msgget()`, `semget()`, `shmget()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/ipc.h>`

### Change History

First released in Issue 4, Version 2.

#### Issue 5

Moved from X/OPEN UNIX extension to BASE.

#### Issue 6

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

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.
NAME
ftruncate — truncate a file to a specified length

SYNOPSIS
#include <unistd.h>
int ftruncate(int fildes, off_t length);

DESCRIPTION
If fildes is not a valid file descriptor open for writing, the ftruncate() function shall fail.
If fildes refers to a regular file, the ftruncate() function shall cause the size of the file to be
truncated to length. If the size of the file previously exceeded length, the extra data shall no
longer be available to reads on the file. If the file previously was smaller than this size, ftruncate() shall either increase the size of the file or fail. XSI-conformant systems shall increase
the size of the file. If the file size is increased, the extended area shall appear as if it were zero-
filled. The value of the seek pointer shall not be modified by a call to ftruncate().

Upon successful completion, if fildes refers to a regular file, the ftruncate() function shall mark
for update the st_ctime and st_mtime fields of the file and the S_ISUID and S_ISGID bits of the file
mode may be cleared. If the ftruncate() function is unsuccessful, the file is unaffected.
If the request would cause the file size to exceed the soft file size limit for the process, the
request shall fail and the implementation shall generate the SIGXFSZ signal for the thread.
If fildes refers to a directory, ftruncate() shall fail.
If fildes refers to any other file type, except a shared memory object, the result is unspecified.
If fildes refers to a shared memory object, ftruncate() shall set the size of the shared memory
object to length.
If the effect of ftruncate() is to decrease the size of a shared memory object or memory mapped
file and whole pages beyond the new end were previously mapped, then the whole pages
beyond the new end shall be discarded.
If the Memory Protection option is supported, references to discarded pages shall result in the
generation of a SIGBUS signal; otherwise, the result of such references is undefined.
If the effect of ftruncate() is to increase the size of a shared memory object, it is unspecified
whether the contents of any mapped pages between the old end-of-file and the new are flushed
to the underlying object.

RETURN VALUE
Upon successful completion, ftruncate() shall return 0; otherwise, -1 shall be returned and errno
set to indicate the error.

ERRORS
The ftruncate() function shall fail if:
[EINTR] A signal was caught during execution.
[EINVAL] The length argument was less than 0.
[EFBIG] or [EINVAL] The length argument was greater than the maximum file size.
XSI [EFBIG] The file is a regular file and length is greater than the offset maximum
established in the open file description associated with fildes.
[EIO] An I/O error occurred while reading from or writing to a file system.
ftruncate()
NAME
ftrylockfile — stdio locking functions

SYNOPSIS

```c
#include <stdio.h>
int ftrylockfile(FILE *file);
```

DESCRIPTION
Refer to `flockfile()`.
NAME
ftw — traverse (walk) a file tree

SYNOPSIS
XSI
#include <ftw.h>
int ftw(const char *path, int (*fn)(const char *,
       const struct stat *ptr, int flag), int ndirs);

DESCRIPTION
The ftw() function shall recursively descend the directory hierarchy rooted in path. For each
object in the hierarchy, ftw() shall call the function pointed to by fn, passing it a pointer to a
null-terminated character string containing the name of the object, a pointer to a stat structure
containing information about the object, and an integer. Possible values of the integer, defined
in the <ftw.h> header, are:

FTW_D For a directory.
FTW_DNR For a directory that cannot be read.
FTW_F For a file.
FTW_SL For a symbolic link (but see also FTW_NS below).
FTW_NS For an object other than a symbolic link on which stat() could not successfully be
executed. If the object is a symbolic link and stat() failed, it is unspecified whether
ftw() passes FTW_SL or FTW_NS to the user-supplied function.

If the integer is FTW_DNR, descendants of that directory shall not be processed. If the integer is
FTW_NS, the stat structure contains undefined values. An example of an object that would
cause FTW_NS to be passed to the function pointed to by fn would be a file in a directory with
read but without execute (search) permission.

The ftw() function shall visit a directory before visiting any of its descendants.

The ftw() function shall use at most one file descriptor for each level in the tree.

The argument ndirs should be in the range [1,OPEN_MAX].

The tree traversal shall continue until either the tree is exhausted, an invocation of fn returns a
non-zero value, or some error, other than [EACCES], is detected within ftw().

The ndirs argument shall specify the maximum number of directory streams or file descriptors
or both available for use by ftw() while traversing the tree. When ftw() returns it shall close any
directory streams and file descriptors it uses not counting any opened by the application-
supplied fn function.

The results are unspecified if the application-supplied fn function does not preserve the current
working directory.

The ftw() function need not be reentrant. A function that is not required to be reentrant is not
required to be thread-safe.

RETURN VALUE
If the tree is exhausted, ftw() shall return 0. If the function pointed to by fn returns a non-zero
value, ftw() shall stop its tree traversal and return whatever value was returned by the function
pointed to by fn. If ftw() detects an error, it shall return −1 and set errno to indicate the error.

If ftw() encounters an error other than [EACCES] (see FTW_DNR and FTW_NS above), it shall
return −1 and set errno to indicate the error. The external variable errno may contain any error
value that is possible when a directory is opened or when one of the \textit{stat} functions is executed on a directory or file.

**ERRORS**

The \texttt{ftw()} function shall fail if:

- **[EACCES]** Search permission is denied for any component of \texttt{path} or read permission is denied for \texttt{path}.
- **[ELOOP]** A loop exists in symbolic links encountered during resolution of the \texttt{path} argument.
- **[ENAMETOOLONG]** The length of the \texttt{path} argument exceeds \{PATH\_MAX\} or a pathname component is longer than \{NAME\_MAX\}.
- **[ENOENT]** A component of \texttt{path} does not name an existing file or \texttt{path} is an empty string.
- **[ENOTDIR]** A component of \texttt{path} is not a directory.
- **[EOVERFLOW]** A field in the \texttt{stat} structure cannot be represented correctly in the current programming environment for one or more files found in the file hierarchy.

The \texttt{ftw()} function may fail if:

- **[EINVAL]** The value of the \texttt{ndirs} argument is invalid.
- **[ELOOP]** More than \{SYMLOOP\_MAX\} symbolic links were encountered during resolution of the \texttt{path} argument.
- **[ENAMETOOLONG]** Pathname resolution of a symbolic link produced an intermediate result whose length exceeds \{PATH\_MAX\}.

In addition, if the function pointed to by \texttt{fn} encounters system errors, \texttt{errno} may be set accordingly.

**EXAMPLES**

**Walking a Directory Structure**

The following example walks the current directory structure, calling the \texttt{fn} function for every directory entry, using at most 10 file descriptors:

```c
#include <ftw.h>
...
if (ftw(".", fn, 10) != 0) {
  perror("ftw"); exit(2);
}
```

**APPLICATION USAGE**

The \texttt{ftw()} function may allocate dynamic storage during its operation. If \texttt{ftw()} is forcibly terminated, such as by \texttt{longjmp()} or \texttt{siglongjmp()} being executed by the function pointed to by \texttt{fn} or an interrupt routine, \texttt{ftw()} does not have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function pointed to by \texttt{fn} return a non-zero value at its next invocation.
RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
longjmp(), lstat(), malloc(), nftw(), opendir(), siglongjmp(), stat(), the Base Definitions volume of IEEE Std 1003.1-2001, <ftw.h>, <sys/stat.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
UX codings in the DESCRIPTION, RETURN VALUE, and ERRORS sections are changed to EX.

Issue 6
The ERRORS section is updated as follows:

• The wording of the mandatory [ELOOP] error condition is updated.
• A second optional [ELOOP] error condition is added.
• The [EOVERFLOW] mandatory error condition is added.

Text is added to the DESCRIPTION to say that the ftw() function need not be reentrant and that the results are unspecified if the application-supplied fn function does not preserve the current working directory.
funlockfile() — stdio locking functions

SYNOPSIS

TSF

```c
#include <stdio.h>

void funlockfile(FILE *file);
```

DESCRIPTION

Refer to `flockfile()`.
NAME
fwide — set stream orientation

SYNOPSIS
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *stream, int mode);

DESCRIPTION
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

The fwide() function shall determine the orientation of the stream pointed to by stream. If mode is
greater than zero, the function first attempts to make the stream wide-oriented. If mode is less
than zero, the function first attempts to make the stream byte-oriented. Otherwise, mode is zero
and the function does not alter the orientation of the stream.

If the orientation of the stream has already been determined, fwide() shall not change it.

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 fwide(), then check errno, and if it is non-zero, assume
an error has occurred.

RETURN VALUE
The fwide() function shall return a value greater than zero if, after the call, the stream has wide-
orientation, a value less than zero if the stream has byte-orientation, or zero if the stream has no
orientation.

ERRORS
The fwide() function may fail if:

 EBADF] The stream argument is not a valid stream.

EXAMPLES
None.

APPLICATION USAGE
A call to fwide() with mode set to zero can be used to determine the current orientation of a
stream.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
(E).

Issue 6
Extensions beyond the ISO C standard are marked.
NAME

fwprintf, swprintf, wprintf — print formatted wide-character output

SYNOPSIS

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

int fwprintf(FILE *restrict stream, const wchar_t *restrict format, ...);
int swprintf(wchar_t *restrict ws, size_t n,
             const wchar_t *restrict format, ...);
int wprintf(const wchar_t *restrict format, ...);
```

DESCRIPTION

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 `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).

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.

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).

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.

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.

In `format` wide-character strings containing the `%` form of conversion specification, each argument in the argument list shall be used exactly once.

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 (`'.'`).

Each conversion specification is introduced by the `'%'` wide character or by the wide-character sequence `"%n$"`, after which the following appear in sequence:
In strings containing the integer argument containing the field width or precision, for example:

\[
\text{wprintf(}L\%1$d:%2$.*3$d:%4$.*3$d\text{\n}}\text{, hour, min, precision, sec);}
\]

15302 - Zero or more flags (in any order), which modify the meaning of the conversion specification.

15303 - An optional minimum field width. If the converted value has fewer wide characters than the 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.

15304 - 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.

15305 - An optional length modifier that specifies the size of the argument.

15306 - A conversion specifier wide character that indicates the type of conversion to be applied. 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 wide-character 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:

\[
\text{wprintf(}L\%1$d:%2$.*3$d:%4$.*3$d\text{\n}}\text{, hour, min, precision, sec);}
\]

The flag wide characters and their meanings are:

15328 - The integer portion of the result of a decimal conversion (%, d, u, f, F, g, or %G) shall be formatted with thousands' grouping wide characters. For other conversions, the behavior is undefined. The numeric grouping wide character is used.

15330 - The result of the conversion shall be left-justified within the field. The conversion shall be right-justified if this flag is not specified.

15333 - 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.

15336 - If the first wide character of a signed conversion is not a sign, or if a signed conversion results in no wide 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.

15339 - 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 0. For x 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 it. 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.
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.

The length modifiers and their meanings are:

- **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.

- **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.

- **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.

- **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.

- **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.

- **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.

- **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.

- **L**: Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `long double` argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

- **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.
The `unsigned` argument shall be converted to unsigned octal format in the style "dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.

The `unsigned` argument shall be converted to unsigned decimal format in the style "dddd". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.

The `unsigned` argument shall be converted to unsigned hexadecimal format in the style "dddd"; the letters "abcdef" are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.

Equivalent to the x conversion specifier, except that letters "ABCDEF" are used instead of "abcdef".

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.

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.

The `double` argument shall be converted in the style "[−]d.ddde±dd", where there shall be one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it shall be equal to the precision; if the precision is missing, it shall be taken as 6; if the precision is zero and no ′#′ flag is present, no radix character shall appear. The value shall be rounded in an implementation-defined manner to the appropriate number of digits. The E conversion wide character shall produce a number with ′E′ instead of ′e′ introducing the exponent. The exponent shall always contain at least two digits. If the value is zero, the exponent shall be zero.

A `double` argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

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.

A `double` argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.
A double argument representing a floating-point number shall be converted in the style "[-]0xh.hhhhp ±d", where there shall be one hexadecimal digit (which is non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point wide character and the number of hexadecimal digits after it shall be equal to the precision; if the precision is missing and FLT_RADIX is a power of 2, then the precision shall be sufficient for an exact representation of the value; if the precision is missing and FLT_RADIX is not a power of 2, then the precision shall be sufficient to distinguish values of type double, except that trailing zeros may be omitted; if the precision is zero and the ‘#’ flag is not specified, no decimal-point wide character shall appear. The letters "abcdef" are used for a conversion and the letters "ABCDEF" for A conversion. The A conversion specifier produces a number with ‘X’ and ‘P’ instead of ‘x’ and ‘p’. The exponent shall always contain at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent shall be zero.

A double argument representing an infinity or NaN shall be converted in the style of an f or F conversion specifier.

If no l (ell) qualifier is present, the int argument shall be converted to a wide character as if by calling the btowc() function and the resulting wide character shall be written. Otherwise, the wint_t argument shall be converted to wchar_t, and written.

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 mbtowc() 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.

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.

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.

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 fprintf() 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.
If a conversion specification does not match one of the above forms, the behavior is undefined.

In no case does 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 `fwprintf()` and `wprintf()` shall be printed as if `fputwc()` had been called.

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.

For 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 significant 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 $L \leq D \leq U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

The `st_ctime` and `st_mtime` fields of the file shall be marked for update between the call to a successful execution of `fwprintf()` or `wprintf()` and the next successful completion of a call to `fflush()` or `fclose()` on the same stream, or a call to `exit()` or `abort()`.

Upon successful completion, these functions shall return the number of wide characters transmitted, excluding the terminating null wide character in the case of `swprintf()`, or a negative value if an output error was encountered, and set `errno` to indicate the error.

If $n$ or more wide characters were requested to be written, `swprintf()` shall return a negative value, and set `errno` to indicate the error.

For the conditions under which `fwprintf()` and `wprintf()` fail and may fail, refer to `fputwc()`.

In addition, all forms of `fwprintf()` may fail if:

- `[EILSEQ]` A wide-character code that does not correspond to a valid character has been detected.
- `[EINVAL]` There are insufficient arguments.

In addition, `wprintf()` and `fwprintf()` may fail if:

- `[ENOMEM]` Insufficient storage space is available.

To print the language-independent date and time format, the following statement could be used:

```c
wprintf(format, weekday, month, day, hour, min);
```

For American usage, `format` could be a pointer to the wide-character string:

```c
L"%s, %s %d, %d:%.2d
"
```

producing the message:

```
Sunday, July 3, 10:02
```

whereas for German usage, `format` could be a pointer to the wide-character string:

```c
L"%s %s %d %d:%.2d
"
```
fwprintf(
)

producing the message:
Sonntag, 3. Juli, 10:02

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
btowc(), fputwc(), fscanff(), mbtowc(), setlocale(), the Base Definitions volume of
IEEE Std 1003.1-2001, Chapter 7, Locale, <stdio.h>, <wchar.h>

CHANGE HISTORY
(E).

Issue 6
The Open Group Corrigendum U040/1 is applied to the RETURN VALUE section, describing
the case if n or more wide characters are requested to be written using swprintf().

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

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

• The prototypes for fwprintf(), swprintf(), and vprintf() are updated.
• The DESCRIPTION is updated.
• The hh, ll, j, t, and z length modifiers are added.
• The a, A, and F conversion characters are added.
• XSI shading is removed from the description of character string representations of infinity
and NaN floating-point values.

The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion
specification” consistently.

NAME
fwrite — binary output

SYNOPSIS
#include <stdio.h>

size_t fwrite(const void *restrict ptr, size_t size, size_t nitems,
              FILE *restrict stream);

DESCRIPTION
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

The fwrite() function shall write, from the array pointed to by ptr, up to nitems elements whose
size is specified by size, to the stream pointed to by stream. For each object, size calls shall be
made to the fputc() function, taking the values (in order) from 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 written. If an error occurs, the resulting value of the file-
position indicator for the stream is unspecified.

The st_ctime and st_mtime fields of the file shall be marked for update between the successful
execution of fwrite() and the next successful completion of a call to fflush() or fclose() on the
same stream, or a call to exit() or abort().

RETURN VALUE
The fwrite() function shall return the number of elements successfully written, which may be
less than nitems if a write error is encountered. If size or nitems is 0, fwrite() shall return 0 and the
state of the stream remains unchanged. Otherwise, if a write error occurs, the error indicator for
the stream shall be set, and errno shall be set to indicate the error.

ERRORS
Refer to fputc().

EXEMPLARY
None.

APPLICATION USAGE
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
or by the same application on a different processor.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
ferror(), fopen(), printf(), putc(), puts(), write(), the Base Definitions volume of
IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:
• The `fwrite()` prototype is updated.
• The DESCRIPTION is updated to clarify how the data is written out using `fputc()`.
NAME
fwscanf, swscanf, wscanf — convert formatted wide-character input

SYNOPSIS
#include <stdio.h>
#include <wchar.h>

int fwscanf(FILE *restrict stream, const wchar_t *restrict format, ...);
int swscanf(const wchar_t *restrict ws, const wchar_t *restrict format, ...);
int wscanf(const wchar_t *restrict format, ...);

DESCRIPTION
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 {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.

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.

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 Nth argument requires that all the leading arguments, from the first to the (N-1)th, are pointers.

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 (’.’).

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:

• An optional assignment-suppressing character ‘*’.
• An optional non-zero decimal integer that specifies the maximum field width.
• An optional length modifier that specifies the size of the receiving object.

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
• A conversion specifier wide character that specifies the type of conversion to be applied. The
valid conversion specifiers are described below.

The `fwscanf()` functions shall execute each directive of the format in turn. If a directive fails, as
detailed below, the function shall return. Failures are described as input failures (due to the
unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space wide characters is executed by reading input
until no more valid input can be read, or up to the first wide character which is not a white-
space wide character, which remains unread.

A directive that is an ordinary wide character shall be executed as follows. The next wide
character is read from the input and compared with the wide character that comprises the
directive; if the comparison shows that they are not equivalent, the directive shall fail, and the
differing and subsequent wide characters remain unread. Similarly, if end-of-file, an encoding
error, or a read error prevents a wide character from being read, the directive shall fail.

A directive that is a conversion specification defines a set of matching input sequences, as
described below for each conversion wide character. A conversion specification is executed in the
following steps.

Input white-space wide characters (as specified by `iswspace()`) shall be skipped, unless the
conversion specification includes a `, c, or n conversion specifier.

An item shall be read from the input, unless the conversion specification includes an n
conversion specifier wide character. An input item is defined as the longest sequence of input
wide characters, not exceeding any specified field width, which is an initial subsequence of a
matching sequence. The first wide character, if any, after the input item shall remain unread. If
the length of the input item is zero, the execution of the conversion specification shall fail; this
condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented
input from the stream, in which case it is an input failure.

Except in the case of a % conversion specifier, the input item (or, in the case of a %n conversion
specification, the count of input wide characters) shall be converted to a type appropriate to the
conversion wide character. If the input item is not a matching sequence, the execution of the
conversion specification shall fail; this condition is a matching failure. Unless assignment
suppression was indicated by a `*` , the result of the conversion shall be placed in the object
pointed to by the first argument following the `format` argument that has not already received a
conversion result if the conversion specification is introduced by %, or in the n th argument if
introduced by the wide-character sequence `"%nS"`. 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.

The length modifiers and their meanings are:

- `hh` Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
  argument with type pointer to `signed char` or `unsigned char`.

- `h` Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
  argument with type pointer to `short` or `unsigned short`.

- `l` (ell) Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an
  argument with type pointer to `long` or `unsigned long`; or that a following c, s, or [ conversion specifier applies to an argument with type pointer to `double`; or that a
  following c, s, or [ conversion specifier applies to an argument with type pointer to
  `wchar_t`. 

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
11 (ell-ell)

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to long long or unsigned long long.

j

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to intmax_t or uintmax_t.

z

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to size_t or the corresponding signed integer type.

t

Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to ptrdiff_t or the corresponding unsigned type.

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.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The following conversion specifier wide characters are valid:

d

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of wcstol() with the value 10 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to int.

i

Matches an optionally signed integer, whose format is the same as expected for the subject sequence of wcstol() with 0 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to int.

o

Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of wcstoul() with the value 8 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to unsigned.

u

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of wcstoul() with the value 10 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to unsigned.

x

Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of wcstoul() with the value 16 for the base argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to unsigned.

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 wcstod(). In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to float.

If the fprintf() 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 fscanf() family of functions shall recognize them as input.

s

Matches a sequence of non white-space wide characters. If no 1 (ell) qualifier is present, characters from the input field shall be converted as if by repeated calls to the wcrtomb() function, with the conversion state described by an mbstate_t object.
initialized to zero before the first wide character is converted. The application shall
ensure that the corresponding argument is a pointer to a character array large enough
to accept the sequence and the terminating null character, which shall be added
automatically.

Otherwise, the application shall ensure that the corresponding argument is a pointer to
an array of `wchar_t` large enough to accept the sequence and the terminating null wide
character, which shall be added automatically.

[ ] Matches a non-empty sequence of wide characters from a set of expected wide
characters (the `scanset`). If no 1 (ell) qualifier is present, wide characters from the input
field shall be converted as if by repeated calls to the `wctomb()` function, with the
conversion state described by an `mbstate_t` object initialized to zero before the first
wide character is converted. The application shall ensure that the corresponding
argument is a pointer to a character array large enough to accept the sequence and the
terminating null character, which shall be added automatically.

If an 1 (ell) qualifier is present, the application shall ensure that the corresponding
argument is a pointer to an array of `wchar_t` large enough to accept the sequence and
the terminating null wide character, which shall be added automatically.

The conversion specification includes all subsequent wide characters in the `format`
string up to and including the matching right square bracket (`']'). The wide
characters between the square brackets (the `scanlist`) comprise the scanset, unless the
wide character after the left square bracket is a circumflex (`'^'`), in which case the
scanset contains all wide characters that do not appear in the scanlist between the
circumflex and the right square bracket. If the conversion specification begins with
"[ ]" or "[^]", the right square bracket is included in the scanlist and the next right
square bracket is the matching right square bracket that ends the conversion
specification; otherwise, the first right square bracket is the one that ends the
conversion specification. If a `'−'` is in the scanlist and is not the first wide character,
nor the second where the first wide character is a `'−'`, nor the last wide character, the
behavior is implementation-defined.

c Matches a sequence of wide characters of exactly the number specified by the field
width (1 if no field width is present in the conversion specification).

If no 1 (ell) length modifier is present, characters from the input field shall be converted
as if by repeated calls to the `wctomb()` function, with the conversion state described by
an `mbstate_t` object initialized to zero before the first wide character is converted. The
corresponding argument shall be a pointer to the initial element of a character array
large enough to accept the sequence. No null character is added.

If an 1 (ell) length modifier is present, the corresponding argument shall be a pointer to
the initial element of an array of `wchar_t` large enough to accept the sequence. No null wide
character is added.

Otherwise, the application shall ensure that the corresponding argument is a pointer to
an array of `wchar_t` large enough to accept the sequence. No null wide character is
added.

p Matches an implementation-defined set of sequences, which shall be the same as the set
of sequences that is produced by the `%p` conversion specification of the corresponding
`fprintf()` functions. The application shall ensure that the corresponding argument is a
pointer to a pointer to `void`. The interpretation of the input item is implementation-
defined. If the input item is a value converted earlier during the same program
execution, the pointer that results shall compare equal to that value; otherwise, the
behavior of the \%p conversion is undefined.

\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 \texttt{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.

\begin{verbatim}
\textbf{XSI}  
\begin{tabular}{l}
\texttt{C} \hspace{1cm} Equivalent to \texttt{lc}. \\
\texttt{S} \hspace{1cm} Equivalent to \texttt{ls}.
\end{tabular}
\end{verbatim}

\% Matches a single \%' wide character; no conversion or assignment shall occur. The complete conversion specification shall be \%.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers \texttt{A}, \texttt{E}, \texttt{F}, \texttt{G}, and \texttt{X} are also valid and shall be equivalent to, respectively, \texttt{a}, \texttt{e}, \texttt{f}, \texttt{g}, and \texttt{x}.

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.

Reaching the end of the string in \texttt{swscanf()} shall be equivalent to encountering end-of-file for \texttt{fwscanf()}.

If conversion terminates on a conflicting input, the offending input shall be left unread in the input. Any trailing white space (including \texttt{<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.

\begin{verbatim}
\textbf{CX}  
\begin{tabular}{l}
\texttt{fscanf()} and \texttt{wscanf()} functions may mark the \texttt{st_atime} field of the file associated with \texttt{stream} for update. The \texttt{st_atime} field shall be marked for update by the first successful execution of \texttt{fgetc()}, \texttt{fgetwc()}, \texttt{fgets()}, \texttt{fgetws()}, \texttt{fread()}, \texttt{getc()}, \texttt{getwc()}, \texttt{getchar()}, \texttt{getwchar()}, \texttt{gets()}, \texttt{fscanf()}, or \texttt{fwscanf()} using \texttt{stream} that returns data not supplied by a prior call to \texttt{ungetc()}.
\end{tabular}
\end{verbatim}

\begin{verbatim}
\textbf{RETURN VALUE}
\end{verbatim}

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 \texttt{errno} shall be set to indicate the error.

\begin{verbatim}
\textbf{ERRORS}
\end{verbatim}

For the conditions under which the \texttt{fwscanf()} functions shall fail and may fail, refer to \texttt{fgetwc()}.

In addition, \texttt{fwscanf()} may fail if:

\begin{verbatim}
\textbf{XSI}  
\begin{tabular}{l}
\texttt{[EILSEQ]} \hspace{1cm} Input byte sequence does not form a valid character.
\texttt{[EINVAL]} \hspace{1cm} There are insufficient arguments.
\end{tabular}
\end{verbatim}
EXAMPLES

The call:

```c
int i, n; float x; char name[50];
n = wscanf(L"%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 Hamster
```

assigns to \( n \) the value 3, to \( i \) the value 25, to \( x \) the value 5.432, and \( name \) contains the string "Hamster".

The call:

```c
int i; float x; char name[50];
(void) wscanf(L"%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

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'.

APPLICATION USAGE

In format strings containing the ‘%’ form of conversion specifications, each argument in the argument list is used exactly once.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`getwc()`, `fwprintf()`, `setlocale()`, `wcstod()`, `wcstol()`, `wcstoul()`, `wcrtomb()`, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, `<langinfo.h>`, `<stdio.h>`, `<wchar.h>`

CHANGE HISTORY


Issue 6

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

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The prototypes for `fwscanf()` and `swscanf()` are updated.
- The DESCRIPTION is updated.
- The `hh`, `ll`, `j`, `t`, and `z` length modifiers are added.
- The `a`, `A`, and `F` conversion characters are added.

The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion specification” consistently.
NAME
gai_strerror — address and name information error description

SYNOPSIS
#include <netdb.h>
const char *gai_strerror(int ecode);

DESCRIPTION
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.

When the ecode argument is one of the following values listed in the <netdb.h> header:

- [EAI_AGAIN]
- [EAI_BADFLAGS]
- [EAI_FAIL]
- [EAI_FAMILY]
- [EAI_MEMORY]
- [EAI_NONAME]
- [EAI_SERVICE]
- [EAI_SOCKTYPE]
- [EAI_SYSTEM]

the function return value shall point to a string describing the error. If the argument is not one of those values, the function shall return a pointer to a string whose contents indicate an unknown error.

RETURN VALUE
Upon successful completion, gai_strerror() shall return a pointer to an implementation-defined string.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getaddrinfo(), the Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>

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

The Open Group Base Resolution bwg2001-009 is applied, which changes the return type from char * to const char *. This is for coordination with the IPnG Working Group.
NAME
gcvt — convert a floating-point number to a string (LEGACY)

SYNOPSIS
#include <stdlib.h>
char *gcvt(double value, int ndigit, char *buf);

DESCRIPTION
Refer to ecvt().
NAME

getaddrinfo — get address information

SYNOPSIS

```c
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *restrict nodename,
    const char *restrict servname,
    const struct addrinfo *restrict hints,
    struct addrinfo **restrict res);
```

DESCRIPTION

Refer to `freeaddrinfo()`.
getc( )

NAME
getc — get a byte from a stream

SYNOPSIS
#include <stdio.h>
int getc(FILE *stream);

DESCRIPTION
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
The getc() function shall be equivalent to fgetc(), except that if it is implemented as a macro it
may evaluate stream more than once, so the argument should never be an expression with side
effects.

RETURN VALUE
Refer to fgetc().

ERRORS
Refer to fgetc().

EXAMPLES
None.

APPLICATION USAGE
If the integer value returned by getc() 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.
Since it may be implemented as a macro, getc() may treat incorrectly a stream argument with
side effects. In particular, getc(’f++’) does not necessarily work as expected. Therefore, use of this
function should be preceded by "#undef getc" in such situations; fgetc() could also be used.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fgetc(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked — stdio with explicit client locking

SYNOPSIS
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);

DESCRIPTION
Versions of the functions getc(), getchar(), putc(), and putchar() respectively named
getc_unlocked(), getchar_unlocked(), putc_unlocked(), and putchar_unlocked() shall be provided
which are functionally equivalent to the original versions, with the exception that they are not
required to be implemented in a thread-safe manner. They may only safely be used within a
scope protected by flockfile() (or ftrylockfile()) and funlockfile(). These functions may safely be
used in a multi-threaded program if and only if they are called while the invoking thread owns
the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions.

RETURN VALUE
See getc(), getchar(), putc(), and putchar().

ERRORS
See getc(), getchar(), putc(), and putchar().

EXAMPLES
None.

APPLICATION USAGE
Since they may be implemented as macros, getc_unlocked() and putc_unlocked() may treat
incorrectly a stream argument with side effects. In particular, getc_unlocked("f++") and
putc_unlocked("f++") do not necessarily work as expected. Therefore, use of these functions in
such situations should be preceded by the following statement as appropriate:

#define getc_unlocked
#define putc_unlocked

RATIONALE
Some I/O functions are typically implemented as macros for performance reasons (for example,
putc() and getc()). For safety, they need to be synchronized, but it is often too expensive to
synchronize on every character. Nevertheless, it was felt that the safety concerns were more
important; consequently, the getc(), getchar(), putc(), and putchar() functions are required to be
thread-safe. However, unlocked versions are also provided with names that clearly indicate the
unsafe nature of their operation but can be used to exploit their higher performance. These
unlocked versions can be safely used only within explicitly locked program regions, using
exported locking primitives. In particular, a sequence such as:

flockfile(fileptr);
putc_unlocked('l', fileptr);
putc_unlocked('
', fileptr);
fprintf(fileptr, "Line 2\n");
funlockfile(fileptr);

is permissible, and results in the text sequence:
getc_unlocked()

being printed without being interspersed with output from other threads.

It would be wrong to have the standard names such as getc(), putc(), and so on, map to the “faster, but unsafe” rather than the “slower, but safe” versions. In either case, you would still want to inspect all uses of getc(), putc(), and so on, by hand when converting existing code.

Choosing the safe bindings as the default, at least, results in correct code and maintains the “atomicity at the function” invariant. To do otherwise would introduce gratuitous synchronization errors into converted code. Other routines that modify the stdio (FILE *) structures or buffers are also safely synchronized.

Note that there is no need for functions of the form getc_locked(), putc_locked(), and so on, since this is the functionality of getc(), putc(), et al. It would be inappropriate to use a feature test macro to switch a macro definition of getc() between getc_locked() and getc_unlocked(), since the ISO C standard requires an actual function to exist, a function whose behavior could not be changed by the feature test macro. Also, providing both the xxx_locked() and xxx_unlocked() forms leads to the confusion of whether the suffix describes the behavior of the function or the circumstances under which it should be used.

Three additional routines, flockfile(), ftrylockfile(), and funlockfile() (which may be macros), are provided to allow the user to delineate a sequence of I/O statements that are executed synchronously.

The ungetc() function is infrequently called relative to the other functions/macros so no unlocked variation is needed.

FUTURE DIRECTIONS

None.

SEE ALSO

getc(), getchar(), putc(), putchar(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY

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

Issue 6

These functions are marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U030/2 is applied, adding APPLICATION USAGE describing how applications should be written to avoid the case when the functions are implemented as macros.
NAME
getchar — get a byte from a stdin stream

SYNOPSIS
#include <stdio.h>
int getchar(void);

DESCRIPTION
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 getchar() function shall be equivalent to getc(stdin).

RETURN VALUE
Refer to fgetc().

ERRORS
Refer to fgetc().

EXAMPLES
None.

APPLICATION USAGE
If the integer value returned by getchar() 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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getc(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
getchar_unlocked — stdio with explicit client locking

SYNOPSIS
#include <stdio.h>

int getchar_unlocked(void);

DESCRIPTION
Refer to getc_unlocked().
getcontext()  

NAME  
getcontext, setcontext — get and set current user context  

SYNOPSIS  
#include <ucontext.h>  
int getcontext(ucontext_t *ucp);  
int setcontext(const ucontext_t *ucp);  

DESCRIPTION  
The getcontext() function shall initialize the structure pointed to by ucp to the current user context of the calling thread. The ucontext_t type that ucp points to defines the user context and includes the contents of the calling thread’s machine registers, the signal mask, and the current execution stack.  
The setcontext() function shall restore the user context pointed to by ucp. A successful call to setcontext() shall not return; program execution resumes at the point specified by the ucp argument passed to setcontext(). The ucp argument should be created either by a prior call to getcontext() or makecontext(), or by being passed as an argument to a signal handler. If the ucp argument was created with getcontext(), program execution continues as if the corresponding call of getcontext() had just returned. If the ucp argument was created with makecontext(), program execution continues with the function passed to makecontext(). When that function returns, the thread shall continue as if after a call to setcontext() with the ucp argument that was input to makecontext(). If the uc_link member of the ucontext_t structure pointed to by the ucp argument is equal to 0, then this context is the main context, and the thread shall exit when this context returns. The effects of passing a ucp argument obtained from any other source are unspecified.  

RETURN VALUE  
Upon successful completion, setcontext() shall not return and getcontext() shall return 0; otherwise, a value of −1 shall be returned.  

ERRORS  
No errors are defined.  

EXAMPLES  
Refer to makecontext().  

APPLICATION USAGE  
When a signal handler is executed, the current user context is saved and a new context is created. If the thread leaves the signal handler via longjmp(), then it is unspecified whether the context at the time of the corresponding setjmp() call is restored and thus whether future calls to getcontext() provide an accurate representation of the current context, since the context restored by longjmp() does not necessarily contain all the information that setcontext() requires. Signal handlers should use siglongjmp() or setcontext() instead.  
Conforming applications should not modify or access the uc_mcontext member of ucontext_t. A conforming application cannot assume that context includes any process-wide static data, possibly including errno. Users manipulating contexts should take care to handle these explicitly when required.  
Use of contexts to create alternate stacks is not defined by this volume of IEEE Std 1003.1-2001.
getcontext()

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
bsd_signal(), makecontext(), setcontext(), setjmp(), sigaction(), sigaltstack(), siglongjmp(),
sigprocmask(), sigsetjmp(), the Base Definitions volume of IEEE Std 1003.1-2001, <ucontext.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Moved from X/OPEN UNIX extension to BASE.

The following sentence was removed from the DESCRIPTION: “If the ucp argument was passed
to a signal handler, program execution continues with the program instruction following the
instruction interrupted by the signal.”
NAME
getcwd — get the pathname of the current working directory

SYNOPSIS
#include <unistd.h>
char *getcwd(char *buf, size_t size);

DESCRIPTION
The getcwd() function shall place an absolute pathname of the current working directory in the
array pointed to by buf, and return buf. The pathname copied to the array shall contain no
components that are symbolic links. The size argument is the size in bytes of the character array
pointed to by the buf argument. If buf is a null pointer, the behavior of getcwd() is unspecified.

RETURN VALUE
Upon successful completion, getcwd() shall return the buf argument. Otherwise, getcwd() shall
return a null pointer and set errno to indicate the error. The contents of the array pointed to by
buf are then undefined.

ERRORS
The getcwd() function shall fail if:
[EINVAL] The size argument is 0.
[ERANGE] The size argument is greater than 0, but is smaller than the length of the
pathname +1.

The getcwd() function may fail if:
[EACCES] Read or search permission was denied for a component of the pathname.
ENOMEM Insufficient storage space is available.

EXAMPLES
Determining the Absolute Pathname of the Current Working Directory
The following example returns a pointer to an array that holds the absolute pathname of the
current working directory. The pointer is returned in the ptr variable, which points to the buf
array where the pathname is stored.

#include <stdlib.h>
#include <unistd.h>
... long size;
char *buf;
char *ptr;
size = pathconf(".", _PC_PATH_MAX);
if ((buf = (char *)malloc((size_t)size)) != NULL)
    ptr = getcwd(buf, (size_t)size);
... APPLICATION USAGE
None.
getcwd()

RATIONALE
Since the maximum pathname length is arbitrary unless \[\text{PATH\_MAX}\] is defined, an application generally cannot supply a \textit{buf} with size \([\text{PATH\_MAX}]+1\).

Having \texttt{getcwd()} take no arguments and instead use the \texttt{malloc()} function to produce space for the returned argument was considered. The advantage is that \texttt{getcwd()} knows how big the working directory pathname is and can allocate an appropriate amount of space. But the programmer would have to use the \texttt{free()} function to free the resulting object, or each use of \texttt{getcwd()} would further reduce the available memory. Also, \texttt{malloc()} and \texttt{free()} are used nowhere else in this volume of IEEE Std 1003.1-2001. Finally, \texttt{getcwd()} is taken from the SVID where it has the two arguments used in this volume of IEEE Std 1003.1-2001.

The older function \texttt{getwd()} was rejected for use in this context because it had only a buffer argument and no size argument, and thus had no way to prevent overwriting the buffer, except to depend on the programmer to provide a large enough buffer.

On some implementations, if \textit{buf} is a null pointer, \texttt{getcwd()} may obtain size bytes of memory using \texttt{malloc()}. In this case, the pointer returned by \texttt{getcwd()} may be used as the argument in a subsequent call to \texttt{free()}. Invoking \texttt{getcwd()} with \textit{buf} as a null pointer is not recommended in conforming applications.

If a program is operating in a directory where some (grand)parent directory does not permit reading, \texttt{getcwd()} may fail, as in most implementations it must read the directory to determine the name of the file. This can occur if search, but not read, permission is granted in an intermediate directory, or if the program is placed in that directory by some more privileged process (for example, login). Including the [EACCESS] error condition makes the reporting of the error consistent and warns the application writer that \texttt{getcwd()} can fail for reasons beyond the control of the application writer or user. Some implementations can avoid this occurrence (for example, by implementing \texttt{getcwd()} using \texttt{pwd}, where \texttt{pwd} is a set-user-root process), thus the error was made optional. Since this volume of IEEE Std 1003.1-2001 permits the addition of other errors, this would be a common addition and yet one that applications could not be expected to deal with without this addition.

FUTURE DIRECTIONS
None.

SEE ALSO
\texttt{malloc()}, the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<unistd.h>}

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [ENOMEM] optional error condition is added.
NAME
getdate — convert user format date and time

SYNOPSIS
#include <time.h>
struct tm *getdate(const char *string);

DESCRIPTION
The getdate() function shall convert a string representation of a date or time into a broken-down
time.

The external variable or macro getdate_err is used by getdate() to return error values.

Templates are used to parse and interpret the input string. The templates are contained in a text
file identified by the environment variable DATEMSK. The DATEMSK variable should be set to
indicate the full pathname of the file that contains the templates. The first line in the template
that matches the input specification is used for interpretation and conversion into the internal
time format.

The following conversion specifications shall be supported:

%  Equivalent to %.
%a  Abbreviated weekday name.
%A  Full weekday name.
%b  Abbreviated month name.
%B  Full month name.
%c  Locale’s appropriate date and time representation.
%C  Century number [00,99]; leading zeros are permitted but not required.
%d  Day of month [01,31]; the leading 0 is optional.
%d  Date as %m/%d/%y.
%e  Equivalent to %d.
%h  Abbreviated month name.
%H  Hour [00,23].
%I  Hour [01,12].
%m  Month number [01,12].
%M  Minute [00,59].
%n  Equivalent to <newline>.
%p  Locale’s equivalent of either AM or PM.
%r  The locale’s appropriate representation of time in AM and PM notation. In the POSIX
locale, this shall be equivalent to %I:%M:%S %p.
%R  Time as %H:%M.
%S  Seconds [00,60]. The range goes to 60 (rather than stopping at 59) to allow positive leap
seconds to be expressed. Since leap seconds cannot be predicted by any algorithm, leap
second data must come from some external source.
getdate()

%t  Equivalent to <tab>.
%T  Time as %H:%M:%S.
%w  Weekday number (Sunday = [0,6]).
%x  Locale’s appropriate date representation.
%x  Locale’s appropriate time representation.
%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.

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.)
%Y  Year as "ccyy" (for example, 2001).
%Z  Timezone name or no characters if no timezone exists. If the timezone supplied by %Z is
not the timezone that getdate() expects, an invalid input specification error shall result.
The getdate() function calculates an expected timezone based on information supplied
to the function (such as the hour, day, and month).

The match between the template and input specification performed by getdate() shall be case-
insensitive.

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 LC_TIME category (see setlocale()).

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 string shall be ignored.

The results are undefined if the conversion specifications %c, %x, and %X include unsupported
conversion specifications.

The following rules apply for converting the input specification into the internal format:

• If %z is being scanned, then getdate() 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 localtime() had been called.
• 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.
• 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.
• If no hour, minute, and second are given, the current hour, minute, and second shall be
assumed.
• 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.

If a conversion specification in the DATESK file does not correspond to one of the conversion
specifications above, the behavior is unspecified.

The getdate() function need not be reentrant. A function that is not required to be reentrant is not
required to be thread-safe.
Upon successful completion, \texttt{getdate()} shall return a pointer to a \textbf{struct tm}. Otherwise, it shall return a null pointer and set \texttt{getdate_err} to indicate the error.

The \texttt{getdate()} function shall fail in the following cases, setting \texttt{getdate_err} to the value shown in the list below. Any changes to \texttt{errno} are unspecified.

1. The \texttt{DATEMSK} environment variable is null or undefined.
2. The template file cannot be opened for reading.
3. Failed to get file status information.
4. The template file is not a regular file.
5. An I/O error is encountered while reading the template file.
6. Memory allocation failed (not enough memory available).
7. There is no line in the template that matches the input.
8. Invalid input specification. For example, February 31; or a time is specified that cannot be represented in a \texttt{time_t} (representing the time in seconds since the Epoch).

The following example shows the possible contents of a template:

\begin{verbatim}
%m
%A %B %d, %Y, %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d, %m, %Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p, %B %dnd
%A den %d. %B %Y %H.%M Uhr
\end{verbatim}

The following are examples of valid input specifications for the template in Example 1:

\begin{verbatim}
getdate("10/1/87 4 PM");
gdate("Friday");
gdate("Friday September 18, 1987, 10:30:30");
gdate("24,9,1986 10:30");
gdate("at monday the 1st of december in 1986");
gdate("run job at 3 PM, december 2nd");
\end{verbatim}

If the \texttt{LC_TIME} category is set to a German locale that includes \texttt{freitag} as a weekday name and \texttt{oktober} as a month name, the following would be valid:

\begin{verbatim}
gdate("freitag den 10. oktober 1986 10.30 Uhr");
\end{verbatim}

The following example shows how local date and time specification can be defined in the template:
getdate() [16328]

System Interfaces

Invocation Line in Template

getcode("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

4. The following examples help to illustrate the above rules assuming that the current date is Mon Sep 22 12:19:47 EDT 1986 and the LC_TIME category is set to the default C locale:

<table>
<thead>
<tr>
<th>Input</th>
<th>Line in Template</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>%a</td>
<td>Mon Sep 22 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>Sun</td>
<td>%a</td>
<td>Sun Sep 28 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>Fri</td>
<td>%a</td>
<td>Fri Sep 26 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>September</td>
<td>%B</td>
<td>Mon Sep 1 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>January</td>
<td>%B</td>
<td>Thu Jan 1 12:19:47 EST 1987</td>
</tr>
<tr>
<td>December</td>
<td>%B</td>
<td>Mon Dec 1 12:19:47 EST 1986</td>
</tr>
<tr>
<td>Sep Mon</td>
<td>%b %a</td>
<td>Mon Sep 1 12:19:47 EDT 1986</td>
</tr>
<tr>
<td>JanFri</td>
<td>%b %a</td>
<td>Fri Jan 2 12:19:47 EST 1987</td>
</tr>
<tr>
<td>DecMon</td>
<td>%b %a</td>
<td>Mon Dec 1 12:19:47 EST 1986</td>
</tr>
<tr>
<td>Jan Wed 1989</td>
<td>%b %a %Y</td>
<td>Wed Jan 4 12:19:47 EST 1989</td>
</tr>
<tr>
<td>Fri 9</td>
<td>%a %H</td>
<td>Fri Sep 26 09:00:00 EDT 1986</td>
</tr>
<tr>
<td>Feb 10:30</td>
<td>%d %H:%S</td>
<td>Sun Feb 1 10:00:30 EST 1987</td>
</tr>
<tr>
<td>10:30</td>
<td>%H:%M</td>
<td>Tue Sep 23 10:30:00 EDT 1986</td>
</tr>
<tr>
<td>13:30</td>
<td>%H:%M</td>
<td>Mon Sep 22 13:30:00 EDT 1986</td>
</tr>
</tbody>
</table>

APPLICATION USAGE

Although historical versions of getdate() did not require that <time.h> declare the external variable getdate_err, this volume of IEEE Std 1003.1-2001 does require it. The standard developers encourage applications to remove declarations of getdate_err and instead incorporate the declaration by including <time.h>.

Applications should use %Y (4-digit years) in preference to %y (2-digit years).

RATIONALE

In standard locales, the conversion specifications %c, %x, and %X do not include unsupported conversion specifiers and so the text regarding results being undefined is not a problem in that case.

FUTURE DIRECTIONS

None.

SEE ALSO

cftime(), localtime(), setlocale(), strftime(), times(), the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The last paragraph of the DESCRIPTION is added.

The %C conversion specification is added, and the exact meaning of the %y conversion specification is clarified in the DESCRIPTION.
A note indicating that this function need not be reentrant is added to the DESCRIPTION.

The %R conversion specification is changed to follow historical practice.

**Issue 6**

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.

The description of %S is updated so that the valid range is [00,60] rather than [00,61].

The DESCRIPTION is updated to refer to conversion specifications instead of field descriptors for consistency with other functions.
NAME
getegid — get the effective group ID

SYNOPSIS
#include <unistd.h>
gid_t getegid(void);

DESCRIPTION
The getegid() function shall return the effective group ID of the calling process.

RETURN VALUE
The getegid() function shall always be successful and no return value is reserved to indicate an
error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
geteuid(), getgid(), getuid(), setegid(), seteuid(), setgid(), setreuid(), setuid(), the Base
Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

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.
NAME
getenv — get value of an environment variable

SYNOPSIS
#include <stdlib.h>
char *getenv(const char *name);

DESCRIPTION
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.

CX The getenv() function shall search the environment of the calling process (see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables) for the environment variable name if it exists and return a pointer to the value of the environment variable. If the specified environment variable cannot be found, a null pointer shall be returned.

CX The application shall ensure that it does not modify the string pointed to by the getenv() function.

CX The string pointed to may be overwritten by a subsequent call to getenv(), setenv(), or unsetenv(), but shall not be overwritten by a call to any other function in this volume of IEEE Std 1003.1-2001.

CX If the application modifies environ or the pointers to which it points, the behavior of getenv() is undefined.

CX The getenv() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE
Upon successful completion, getenv() shall return a pointer to a string containing the value for the specified name. If the specified name cannot be found in the environment of the calling process, a null pointer shall be returned.

CX The return value from getenv() may point to static data which may be overwritten by subsequent calls to getenv(), setenv(), or unsetenv().

XSI On XSI-conformant systems, the return value from getenv() may point to static data which may also be overwritten by subsequent calls to putenv().

ERRORS
No errors are defined.

EXAMPLES

Getting the Value of an Environment Variable
The following example gets the value of the HOME environment variable.

#include <stdlib.h>
...
const char *name = "HOME";
char *value;
value = getenv(name);
**APPLICATION USAGE**

None.

**RATIONALE**

The `clearenv()` function was considered but rejected. The `putenv()` function has now been included for alignment with the Single UNIX Specification.

The `getenv()` function is inherently not reentrant because it returns a value pointing to static data.

Conforming applications are required not to modify `environ` directly, but to use only the functions described here to manipulate the process environment as an abstract object. Thus, the implementation of the environment access functions has complete control over the data structure used to represent the environment (subject to the requirement that `environ` be maintained as a list of strings with embedded equal signs for applications that wish to scan the environment). This constraint allows the implementation to properly manage the memory it allocates, either by using allocated storage for all variables (copying them on the first invocation of `setenv()` or `unsetenv()`), or keeping track of which strings are currently in allocated space and which are not, via a separate table or some other means. This enables the implementation to free any allocated space used by strings (and perhaps the pointers to them) stored in `environ` when `unsetenv()` is called. A C runtime start-up procedure (that which invokes `main()` and perhaps initializes `environ`) can also initialize a flag indicating that none of the environment has yet been copied to allocated storage, or that the separate table has not yet been initialized.

In fact, for higher performance of `getenv()`, the implementation could also maintain a separate copy of the environment in a data structure that could be searched much more quickly (such as an indexed hash table, or a binary tree), and update both it and the linear list at `environ` when `setenv()` or `unsetenv()` is invoked.

Performance of `getenv()` can be important for applications which have large numbers of environment variables. Typically, applications like this use the environment as a resource database of user-configurable parameters. The fact that these variables are in the user’s shell environment usually means that any other program that uses environment variables (such as `ls`, which attempts to use `COLUMNS`), or really almost any utility (`LANG`, `LC_ALL`, and so on) is similarly slowed down by the linear search through the variables.

An implementation that maintains separate data structures, or even one that manages the memory it consumes, is not currently required as it was thought it would reduce consensus among implementors who do not want to change their historical implementations.

The POSIX Threads Extension states that multi-threaded applications must not modify `environ` directly, and that IEEE Std 1003.1-2001 is providing functions which such applications can use in the future to manipulate the environment in a thread-safe manner. Thus, moving away from application use of `environ` is desirable from that standpoint as well.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`exec`, `putenv()`, `setenv()`, `unsetenv()`, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables, `<stdlib.h>`

**CHANGE HISTORY**

First released in Issue 1. Derived from Issue 1 of the SVID.
**Issue 5**

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

**Issue 6**

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

- References added to the new `setenv()` and `unsetenv()` functions.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
geteuid — get the effective user ID

SYNOPSIS
#include <unistd.h>
uid_t geteuid(void);

DESCRIPTION
The geteuid() function shall return the effective user ID of the calling process.

RETURN VALUE
The geteuid() function shall always be successful and no return value is reserved to indicate an
error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getegid(), getgid(), getuid(), setegid(), seteuid(), setgid(), setregid(), setreuid(), setuid(), the Base
Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

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.
NAME
getgid — get the real group ID

SYNOPSIS
#include <unistd.h>
gid_t getgid(void);

DESCRIPTION
The getgid() function shall return the real group ID of the calling process.

RETURN VALUE
The getgid() function shall always be successful and no return value is reserved to indicate an error.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getegid(), geteuid(), geteuid(), setegid(), seteuid(), setgid(), setreuid(), setregid(), setreuid(), setuid(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

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.
getgrent()

NAME
getgrent — get the group database entry

SYNOPSIS
XSI

```c
#include <grp.h>

struct group *getgrent(void);
```

DESCRIPTION
Refer to endgrent().
NAME
getgrgid, getgrgid_r — get group database entry for a group ID

SYNOPSIS
#include <grp.h>

struct group *getgrgid(gid_t gid);

int getgrgid_r(gid_t gid, struct group *grp, char *buffer,
    size_t bufsize, struct group **result);

DESCRIPTION
The getgrgid() function shall search the group database for an entry with a matching gid.
The getgrgid() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.
The getgrgid_r() function shall update the group structure pointed to by grp and store a pointer to that structure at the location pointed to by result. The structure shall contain an entry from the group database with a matching gid. Storage referenced by the group structure is allocated from the memory provided with the buffer parameter, which is bufsize bytes in size. The maximum size needed for this buffer can be determined with the _SC_GETGR_R_SIZE_MAX sysconf() parameter. A NULL pointer shall be returned at the location pointed to by result on error or if the requested entry is not found.

RETURN VALUE
Upon successful completion, getgrgid() shall return a pointer to a struct group with the structure defined in <grp.h> with a matching entry if one is found. The getgrgid() function shall return a null pointer if either the requested entry was not found, or an error occurred. On error, errno shall be set to indicate the error.
The return value may point to a static area which is overwritten by a subsequent call to getgrent(), getgrgid(), or getgrnam().
If successful, the getgrgid_r() function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS
The getgrgid() and getgrgid_r() functions may fail if:

- [EIO] An I/O error has occurred.
- [EINTR] A signal was caught during getgrgid().
- [EMFILE] (OPEN_MAX) file descriptors are currently open in the calling process.
- [ENFILE] The maximum allowable number of files is currently open in the system.

The getgrgid_r() function may fail if:

- [ERANGE] Insufficient storage was supplied via buffer and bufsize to contain the data to be referenced by the resulting group structure.
getgrgid()

EXAMPLES

Finding an Entry in the Group Database

The following example uses getgrgid() to search the group database for a group ID that was previously stored in a stat structure, then prints out the group name if it is found. If the group is not found, the program prints the numeric value of the group for the entry.

```c
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
...
struct stat statbuf;
struct group *grp;
...
if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %-8.8s", grp->gr_name);
else
    printf(" %-8d", statbuf.st_gid);
...
```

APPLICATION USAGE

Applications wishing to check for error situations should set errno to 0 before calling getgrgid(). If errno is set on return, an error occurred.

The getgrgid_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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endgrent(), getgrnam(), the Base Definitions volume of IEEE Std 1003.1-2001, <grp.h>, <limits.h>, <sys/types.h>

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The getgrgid_r() function is included for alignment with the POSIX Threads Extension.

A note indicating that the getgrgid() function need not be reentrant is added to the DESCRIPTION.

Issue 6

The getgrgid_r() function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION describing matching the gid.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

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 RETURN VALUE section, the requirement to set `errno` on error is added.

- The `[EIO]`, `[EINTR]`, `[EMFILE]`, and `[ENFILE]` optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from `bufsize` characters to bytes.
NAME
getgrnm, getgrnm_r — search group database for a name

SYNOPSIS
#include <grp.h>

struct group *getgrnm(const char *name);
int getgrnm_r(const char *name, struct group *grp, char *buffer,
size_t bufsize, struct group **result);

DESCRIPTION
The getgrnm() function shall search the group database for an entry with a matching name.
The getgrnm() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The getgrnm_r() function shall update the group structure pointed to by grp and store a pointer to that structure at the location pointed to by result. The structure shall contain an entry from the group database with a matching gid or name. Storage referenced by the group structure is allocated from the memory provided with the buffer parameter, which is bufsize bytes in size. The maximum size needed for this buffer can be determined with the _SC_GETGR_R_SIZE_MAX sysconf() parameter. A NULL pointer is returned at the location pointed to by result on error or if the requested entry is not found.

RETURN VALUE
The getgrnm() function shall return a pointer to a struct group with the structure defined in <grp.h> with a matching entry if one is found. The getgrnm() function shall return a null pointer if either the requested entry was not found, or an error occurred. On error, errno shall be set to indicate the error.

The return value may point to a static area which is overwritten by a subsequent call to getgrent(), getgrgid(), or getgrnm().

If successful, the getgrnm_r() function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS
The getgrnm() and getgrnm_r() functions may fail if:

[EIO] An I/O error has occurred.
[EINTR] A signal was caught during getgrnm().
[EMFILE] [OPEN_MAX] file descriptors are currently open in the calling process.
[ENFILE] The maximum allowable number of files is currently open in the system.

The getgrnm_r() function may fail if:

[ERANGE] Insufficient storage was supplied via buffer and bufsize to contain the data to be referenced by the resulting group structure.
EXAMPLES
None.

APPLICATION USAGE
Applications wishing to check for error situations should set errno to 0 before calling getgrnam().
If errno is set on return, an error occurred.
The getgrnam_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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
echgrent(), getgrgid(), the Base Definitions volume of IEEE Std 1003.1-2001, <grp.h>, <limits.h>,<sys/types.h>

CHANGE HISTORY
First released in Issue 1. Derived from System V Release 2.0.

Issue 5
Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.
The getgrnam_r() function is included for alignment with the POSIX Threads Extension.
A note indicating that the getgrnam() function need not be reentrant is added to the DESCRIPTION.

Issue 6
The getgrnam_r() function is marked as part of the Thread-Safe Functions option.
In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
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 RETURN VALUE section, the requirement to set errno on error is added.
• The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.
The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.
IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from bufsize characters to bytes.
**NAME**
getgroups — get supplementary group IDs

**SYNOPSIS**
```c
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
```

**DESCRIPTION**
The `getgroups()` function shall fill in the array `grouplist` with the current supplementary group IDs of the calling process. It is implementation-defined whether `getgroups()` also returns the effective group ID in the `grouplist` array.

The `gidsetsize` argument specifies the number of elements in the array `grouplist`. The actual number of group IDs stored in the array shall be returned. The values of array entries with indices greater than or equal to the value returned are undefined.

If `gidsetsize` is 0, `getgroups()` shall return the number of group IDs that it would otherwise return without modifying the array pointed to by `grouplist`.

If the effective group ID of the process is returned with the supplementary group IDs, the value returned shall always be greater than or equal to one and less than or equal to the value of `{NGROUPS_MAX}+1`.

**RETURN VALUE**
Upon successful completion, the number of supplementary group IDs shall be returned. A return value of −1 indicates failure and `errno` shall be set to indicate the error.

**ERRORS**
The `getgroups()` function shall fail if:

[EINVAL] The `gidsetsize` argument is non-zero and less than the number of group IDs that would have been returned.

**EXAMPLES**

**Getting the Supplementary Group IDs of the Calling Process**
The following example places the current supplementary group IDs of the calling process into the `group` array.
```c
#include <sys/types.h>
#include <unistd.h>
...

gid_t *group;
int ngroups;
long ngroups_max;

ngroups_max = sysconf(_SC_NGROUPS_MAX) + 1;
group = (gid_t *)malloc(ngroups_max *sizeof(gid_t));
ngroups = getgroups(ngroups_max, group);
```

**APPLICATION USAGE**
None.

**RATIONALE**
The related function `setgroups()` is a privileged operation and therefore is not covered by this 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.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`getegid()`, `setgid()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/types.h>`,
`<unistd.h>`

**CHANGE HISTORY**

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

**Issue 5**

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

**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.

- A return value of 0 is not permitted, because `{NGROUPS_MAX}` cannot be 0. This is a FIPS
  requirement.

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

- Explanation added that the effective group ID may be included in the supplementary group
  list.
NAME
gethostbyaddr, gethostbyname — network host database functions

SYNOPSIS
#include <netdb.h>

struct hostent *gethostbyaddr(const void *addr, socklen_t len,
   int type);

struct hostent *gethostbyname(const char *name);

DESCRIPTION
These functions shall retrieve information about hosts. This information is considered to be stored in a database that can be accessed sequentially or randomly. Implementation of this database is unspecified.

Note: In many cases it is implemented by the Domain Name System, as documented in RFC 1034, RFC 1035, and RFC 1886.

Entries shall be returned in hostent structures.

The gethostbyaddr() function shall return an entry containing addresses of address family type for the host with address addr. The len argument contains the length of the address pointed to by addr. The gethostbyaddr() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The gethostbyname() function shall return an entry containing addresses of address family AF_INET for the host with name name. The gethostbyname() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The addr argument of gethostbyaddr() shall be an in_addr structure when type is AF_INET. It contains a binary format (that is, not null-terminated) address in network byte order. The gethostbyaddr() function is not guaranteed to return addresses of address families other than AF_INET, even when such addresses exist in the database.

If gethostbyaddr() returns successfully, then the h_addrtype field in the result shall be the same as the type argument that was passed to the function, and the h_addr_list field shall list a single address that is a copy of the addr argument that was passed to the function.

The name argument of gethostbyname() shall be a node name; the behavior of gethostbyname() when passed a numeric address string is unspecified. For IPv4, a numeric address string shall be in the dotted-decimal notation described in inet_addr().

If name is not a numeric address string and is an alias for a valid host name, then gethostbyname() shall return information about the host name to which the alias refers, and name shall be included in the list of aliases returned.

RETURN VALUE
Upon successful completion, these functions shall return a pointer to a hostent structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found.

Upon unsuccessful completion, gethostbyaddr() and gethostbyname() shall set h_errno to indicate the error.

ERRORS
These functions shall fail in the following cases. The gethostbyaddr() and gethostbyname() functions shall set h_errno to the value shown in the list below. Any changes to errno are unspecified.
gethostbyaddr()

[HOST_NOT_FOUND]  No such host is known.

[NO_DATA]  The server recognized the request and the name, but no address is available.
[NO_DATA]  Another type of request to the name server for the domain might return an answer.

[NO_RECOVERY]  An unexpected server failure occurred which cannot be recovered.

[TRY_AGAIN]  A temporary and possibly transient error occurred, such as a failure of a server to respond.

EXAMPLES

APPLICATION USAGE

The `gethostbyaddr()` and `gethostbyname()` functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

The `getaddrinfo()` and `getnameinfo()` functions are preferred over the `gethostbyaddr()` and `gethostbyname()` functions.

RATIONALE

None.

FUTURE DIRECTIONS

The `gethostbyaddr()` and `gethostbyname()` functions may be withdrawn in a future version.

SEE ALSO

`endhostent()`, `endservent()`, `gai_strerror()`, `getaddrinfo()`, `h_errno`, `inet_addr()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<netdb.h>`

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
gethostent — network host database functions

SYNOPSIS
#include <netdb.h>
struct hostent *gethostent(void);

DESCRIPTION
Refer to endhostent().
gethostid( )

NAME
gethostid — get an identifier for the current host

SYNOPSIS
XSI
#include <unistd.h>
long gethostid(void);

DESCRIPTION
The gethostid() function shall retrieve a 32-bit identifier for the current host.

RETURN VALUE
Upon successful completion, gethostid() shall return an identifier for the current host.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
This volume of IEEE Std 1003.1-2001 does not define the domain in which the return value is unique.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
random(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
NAME
gethostname — get name of current host

SYNOPSIS
#include <unistd.h>
int gethostname(char *name, size_t namelen);

DESCRIPTION
The gethostname() function shall return the standard host name for the current machine. The
name 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.

Host names are limited to {HOST_NAME_MAX} bytes.

RETURN VALUE
Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
gethostid(), uname(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
The Open Group Base Resolution bwg2001-008 is applied, changing the namelen parameter from
socklen_t to size_t.
NAME
getitimer, setitimer — get and set value of interval timer

SYNOPSIS
XSI
#include <sys/time.h>

int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *restrict value,
struct itimerval *restrict ovalue);

DESCRIPTION
The getitimer() function shall store the current value of the timer specified by which into the
structure pointed to by value. The setitimer() function shall set the timer specified by which to
the value specified in the structure pointed to by value, and if ovalue is not a null pointer, store
the previous value of the timer in the structure pointed to by ovalue.

A timer value is defined by the itimerval structure, specified in <sys/time.h>. If it_value is non-
zero, it shall indicate the time to the next timer expiration. If it_interval is non-zero, it shall
specify a value to be used in reloading it_value when the timer expires. Setting it_value to 0 shall
disable a timer, regardless of the value of it_interval. Setting it_interval to 0 shall disable a timer
after its next expiration (assuming it_value is non-zero).

Implementations may place limitations on the granularity of timer values. For each interval
timer, if the requested timer value requires a finer granularity than the implementation supports,
the actual timer value shall be rounded up to the next supported value.

An XSI-conforming implementation provides each process with at least three interval timers,
which are indicated by the which argument:

ITIMER_REAL Decrements in real time. A SIGALRM signal is delivered when this timer
expires.

ITIMER_VIRTUAL Decrements in process virtual time. It runs only when the process is
executing. A SIGVTALRM signal is delivered when it expires.

ITIMER_PROF Decrements both in process virtual time and when the system is running
on behalf of the process. It is designed to be used by interpreters in
statistically profiling the execution of interpreted programs. Each time the
ITIMER_PROF timer expires, the SIGPROF signal is delivered.

The interaction between setitimer() and any of alarm(), sleep(), or usleep() is unspecified.

RETURN VALUE
Upon successful completion, getitimer() or setitimer() shall return 0; otherwise, −1 shall be
returned and errno set to indicate the error.

ERRORS
The setitimer() function shall fail if:

[EINVAL] The value argument is not in canonical form. (In canonical form, the number of
microseconds is a non-negative integer less than 1 000 000 and the number of
seconds is a non-negative integer.)

The getitimer() and setitimer() functions may fail if:

[EINVAL] The which argument is not recognized.
getitimer()

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
alarm(), sleep(), timer_getoverrun(), ualarm(), usleep(), the Base Definitions volume of
IEEE Std 1003.1-2001, <signal.h>, <sys/time.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Moved from X/OPEN UNIX extension to BASE.

The restrict keyword is added to the setitimer() prototype for alignment with the
NAME
getlogin, getlogin_r — get login name

SYNOPSIS
#include <unistd.h>

char *getlogin(void);

int getlogin_r(char *name, size_t namesize);

DESCRIPTION
The getlogin() function shall return a pointer to a string containing the user name associated by
the login activity with the controlling terminal of the current process. If getlogin() returns a non-
null pointer, then that pointer points to the name that the user logged in under, even if there are
several login names with the same user ID.

The getlogin() function need not be reentrant. A function that is not required to be reentrant is
not required to be thread-safe.

The getlogin_r() function shall put the name associated by the login activity with the controlling
terminal of the current process in the character array pointed to by name. The array is namesize
characters long and should have space for the name and the terminating null character. The
maximum size of the login name is {LOGIN_NAME_MAX}.

If getlogin_r() is successful, name points to the name the user used at login, even if there are
several login names with the same user ID.

RETURN VALUE
Upon successful completion, getlogin() shall return a pointer to the login name or a null pointer
if the user’s login name cannot be found. Otherwise, it shall return a null pointer and set errno to
indicate the error.

The return value from getlogin() may point to static data whose content is overwritten by each
call.

If successful, the getlogin_r() function shall return zero; otherwise, an error number shall be
returned to indicate the error.

ERRORS
The getlogin() and getlogin_r() functions may fail if:

[EMFILE] [OPEN_MAX] file descriptors are currently open in the calling process.
[ENFILE] The maximum allowable number of files is currently open in the system.
[ENXIO] The calling process has no controlling terminal.

The getlogin_r() function may fail if:

[ERANGE] The value of namesize is smaller than the length of the string to be returned
including the terminating null character.
EXAMPLES

Getting the User Login Name

The following example calls the `getlogin()` function to obtain the name of the user associated with the calling process, and passes this information to the `getpwnam()` function to get the associated user database information.

```c
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
...
char *lgn;
struct passwd *pw;
...
if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
    fprintf(stderr, "Get of user information failed.\n"); exit(1);
}
```

APPLICATION USAGE

Three names associated with the current process can be determined: `getpwuid(geteuid())` shall return the name associated with the effective user ID of the process; `getlogin()` shall return the name associated with the current login activity; and `getpwuid(getuid())` shall return the name associated with the real user ID of the process.

The `getlogin_r()` function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

The `getlogin()` function returns a pointer to the user’s login name. The same user ID may be shared by several login names. If it is desired to get the user database entry that is used during login, the result of `getlogin()` should be used to provide the argument to the `getpwnam()` function. (This might be used to determine the user’s login shell, particularly where a single user has multiple login shells with distinct login names, but the same user ID.)

The information provided by the `cuserid()` function, which was originally defined in the POSIX.1-1988 standard and subsequently removed, can be obtained by the following:

```c
getpwuid(geteuid())
```

while the information provided by historical implementations of `cuserid()` can be obtained by:

```c
getpwuid(getuid())
```

The thread-safe version of this function places the user name in a user-supplied buffer and returns a non-zero value if it fails. The non-thread-safe version may return the name in a static data area that may be overwritten by each call.

FUTURE DIRECTIONS

None.

SEE ALSO

`getpwnam()`, `getpwuid()`, `geteuid()`, `getuid()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<limits.h>`, `<unistd.h>`
getlogin(

**CHANGE HISTORY**

First released in Issue 1. Derived from System V Release 2.0.

**Issue 5**

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The `getlogin_r` function is included for alignment with the POSIX Threads Extension.

A note indicating that the `getlogin` function need not be reentrant is added to the DESCRIPTION.

**Issue 6**

The `getlogin_r` function is marked as part of the Thread-Safe Functions option.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

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

- In the RETURN VALUE section, the requirement to set `errno` on error is added.
- The [EMFILE], [ENFILE], and [ENXIO] optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.
NAME
getmsg, getpmsg — receive next message from a STREAMS file (STREAMS)

SYNOPSIS

```c
#include <stropts.h>

int getmsg(int fildes, struct strbuf *restrict ctlptr,
            struct strbuf *restrict dataptr, int *restrict flagsp);

int getpmsg(int fildes, struct strbuf *restrict ctlptr,
            struct strbuf *restrict dataptr, int *restrict bandp,
            int *restrict flagsp);
```

DESCRIPTION

The `getmsg()` function shall retrieve the contents of a message located at the head of the
STREAM head read queue associated with a STREAMS file and place the contents into one or
more buffers. The message contains either a data part, a control part, or both. The data and
control parts of the message shall be placed into separate buffers, as described below. The
semantics of each part are defined by the originator of the message.

The `getpmsg()` function shall be equivalent to `getmsg()`, except that it provides finer control over
the priority of the messages received. Except where noted, all requirements on `getmsg()` also
pertain to `getpmsg()`.

The `fildes` argument specifies a file descriptor referencing a STREAMS-based file.

The `ctlptr` and `dataptr` arguments each point to a `strbuf` structure, in which the `buf` member points
to a buffer in which the data or control information is to be placed, and the `maxlen` member
indicates the maximum number of bytes this buffer can hold. On return, the `len` member shall
contain the number of bytes of data or control information actually received. The `len` member
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
control information is present in the message.

When `getmsg()` is called, `flagsp` should point to an integer that indicates the type of message the
process is able to receive. This is described further below.

The `ctlptr` argument is used to hold the control part of the message, and `dataptr` is used to hold
the data part of the message. If `ctlptr` (or `dataptr`) is a null pointer or the `maxlen` member is −1, the
control (or data) part of the message shall not be processed and shall be left on the STREAM
head read queue, and if the `ctlptr` (or `dataptr`) is not a null pointer, `len` shall be set to −1. If the
`maxlen` member is set to 0 and there is a zero-length control (or data) part, that zero-length part
shall be removed from the read queue and `len` shall be set to 0. If the `maxlen` member is set to 0
and there are more than 0 bytes of control (or data) information, that information shall be left on
the read queue and `len` shall be set to 0. If the `maxlen` member in `ctlptr` (or `dataptr`) is less than the
control (or data) part of the message, `maxlen` bytes shall be retrieved. In this case, the remainder
of the message shall be left on the STREAM head read queue and a non-zero return value shall
be provided.

By default, `getmsg()` shall process the first available message on the STREAM head read queue.
However, a process may choose to retrieve only high-priority messages by setting the integer
pointed to by `flagsp` to RS_HIPRI. In this case, `getmsg()` shall only process the next message if it is
a high-priority message. When the integer pointed to by `flagsp` is 0, any available message shall
be retrieved. In this case, on return, the integer pointed to by `flagsp` shall be set to RS_HIPRI if a
high-priority message was retrieved, or 0 otherwise.

For `getpmsg()`, the flags are different. The `flagsp` argument points to a bitmask with the following
mutually-exclusive flags defined: MSG_HIPRI, MSG_BAND, and MSG_ANY. Like `getmsg()`,
getpmsg() shall process the first available message on the STREAM head read queue. A process may choose to retrieve only high-priority messages by setting the integer pointed to by flagsp to MSG_HIPRI and the integer pointed to by bandp to 0. In this case, getpmsg() shall only process the next message if it is a high-priority message. In a similar manner, a process may choose to retrieve a message from a particular priority band by setting the integer pointed to by flagsp to MSG_BAND and the integer pointed to by bandp to the priority band of interest. In this case, getpmsg() shall only process the next message if it is in a priority band equal to, or greater than, the integer pointed to by bandp, or if it is a high-priority message. If a process wants to get the first message off the queue, the integer pointed to by flagsp should be set to MSG_ANY and the integer pointed to by bandp should be set to 0. On return, if the message retrieved was a high-priority message, the integer pointed to by flagsp shall be set to MSG_HIPRI and the integer pointed to by bandp shall be set to 0. Otherwise, the integer pointed to by flagsp shall be set to MSG_BAND and the integer pointed to by bandp shall be set to the priority band of the message.

If O_NONBLOCK is not set, getmsg() and getpmsg() shall block until a message of the type specified by flagsp is available at the front of the STREAM head read queue. If O_NONBLOCK is set and a message of the specified type is not present at the front of the read queue, getmsg() and getpmsg() shall fail and set errno to [EAGAIN].

If a hangup occurs on the STREAM from which messages are retrieved, getmsg() and getpmsg() shall continue to operate normally, as described above, until the STREAM head read queue is empty. Thereafter, they shall return 0 in the len members of ctlptr and dataptr.

RETURN VALUE

Upon successful completion, getmsg() and getpmsg() shall return a non-negative value. A value of 0 indicates that a full message was read successfully. A return value of MORECTL indicates that more control information is waiting for retrieval. A return value of MOREDATA indicates that more data is waiting for retrieval. A return value of the bitwise-logical OR of MORECTL and MOREDATA indicates that both types of information remain. Subsequent getmsg() and getpmsg() calls shall retrieve the remainder of the message. However, if a message of higher priority has come in on the STREAM head read queue, the next call to getmsg() or getpmsg() shall retrieve that higher-priority message before retrieving the remainder of the previous message.

If the high priority control part of the message is consumed, the message shall be placed back on the queue as a normal message of band 0. Subsequent getmsg() and getpmsg() calls shall retrieve the remainder of the message. If, however, a priority message arrives or already exists on the STREAM head, the subsequent call to getmsg() or getpmsg() shall retrieve the higher-priority message before retrieving the remainder of the message that was put back.

Upon failure, getmsg() and getpmsg() shall return −1 and set errno to indicate the error.

ERRORS

The getmsg() and getpmsg() functions shall fail if:

[EAGAIN] The O_NONBLOCK flag is set and no messages are available.

[EBADF] The fildes argument is not a valid file descriptor open for reading.

[EBADMSG] The queued message to be read is not valid for getmsg() or getpmsg() or a pending file descriptor is at the STREAM head.

[EINTR] A signal was caught during getmsg() or getpmsg().

[EINVAL] An illegal value was specified by flagsp, or the STREAM or multiplexer referenced by fildes is linked (directly or indirectly) downstream from a multiplexer.
getmsg()

A STREAM is not associated with fildes.

In addition, getmsg() and getpmsg() shall fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of getmsg() or getpmsg() but reflects the prior error.

EXAMPLES

Getting Any Message

In the following example, the value of fd is assumed to refer to an open STREAMS file. The call to getmsg() retrieves any available message on the associated STREAMS file read queue, returning control and data information to the buffers pointed to by ctrlbuf and databuf, respectively.

```c
#include <stropts.h>
...
int fd;
char ctrlbuf[128];
char databuf[512];
struct strbuf ctrl;
struct strbuf data;
int flags = 0;
int ret;

ctrl.buf = ctrlbuf;
ctrl.maxlen = sizeof(ctrlbuf);
data.buf = databuf;
data.maxlen = sizeof(databuf);
ret = getmsg(fd, &ctrl, &data, &flags);
```

Getting the First Message off the Queue

In the following example, the call to getpmsg() retrieves the first available message on the associated STREAMS file read queue.

```c
#include <stropts.h>
...
int fd;
char ctrlbuf[128];
char databuf[512];
struct strbuf ctrl;
struct strbuf data;
int band = 0;
int flags = MSG_ANY;
int ret;

ctrl.buf = ctrlbuf;
ctrl.maxlen = sizeof(ctrlbuf);
data.buf = databuf;
data.maxlen = sizeof(databuf);
ret = getpmsg(fd, &ctrl, &data, &band, &flags);
```
getmsg()

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.6 (on page 38), poll(), putmsg(), read(), write(), the Base Definitions volume of IEEE Std 1003.1-2001, <stropts.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
A paragraph regarding ‘‘high-priority control parts of messages’’ is added to the RETURN VALUE section.

Issue 6
This function is marked as part of the XSI STREAMS Option Group.
The restrict keyword is added to the getmsg() and getpmsg() prototypes for alignment with the ISO/IEC 9899:1999 standard.
NAME

getnameinfo — get name information

SYNOPSIS

```c
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *restrict sa, socklen_t salen,
    char *restrict node, socklen_t nodelen, char *restrict service,
    socklen_t servicelen, unsigned flags);
```

DESCRIPTION

The getnameinfo() function shall translate a socket address to a node name and service location, all of which are defined as in getaddrinfo().

The sa argument points to a socket address structure to be translated.

If the socket address structure contains an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address, the implementation shall extract the embedded IPv4 address and lookup the node name for that IPv4 address.

Note: The IPv6 unspecified address ("::") and the IPv6 loopback address ("::1") are not IPv4-compatible addresses. If the address is the IPv6 unspecified address ("::"), a lookup is not performed, and the [EAI_NONAME] error is returned.

If the node argument is non-NULL and the nodelen argument is non-zero, then the node argument points to a buffer able to contain up to nodelen characters that receives the node name as a null-terminated string. If the node argument is NULL or the nodelen argument is zero, the node name shall not be returned. If the node’s name cannot be located, the numeric form of the node’s address is returned instead of its name.

If the service argument is non-NULL and the servicelen argument is non-zero, then the service argument points to a buffer able to contain up to servicelen bytes that receives the service name as a null-terminated string. If the service argument is NULL or the servicelen argument is zero, the service name shall not be returned. If the service’s name cannot be located, the numeric form of the service address (for example, its port number) shall be returned instead of its name.

The flags argument is a flag that changes the default actions of the function. By default the fully-qualified domain name (FQDN) for the host shall be returned, but:

- If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN shall be returned for local hosts.
- If the flag bit NI_NUMERICHOST is set, the numeric form of the host’s address shall be returned instead of its name, under all circumstances.
- If the flag bit NI_NAMEREQD is set, an error shall be returned if the host’s name cannot be located.
- If the flag bit NI_NUMERICSERV is set, the numeric form of the service address shall be returned (for example, its port number) instead of its name, under all circumstances.
- If the flag bit NI_DGRAM is set, this indicates that the service is a datagram service (SOCK_DGRAM). The default behavior shall assume that the service is a stream service (SOCK_STREAM).

Notes:

1. The two NI_NUMERICxxx flags are required to support the –n flag that many commands provide.
getnameinfo()

2. The NI_DGRAM flag is required for the few AF_INET and AF_INET6 port numbers (for example, [512,514]) that represent different services for UDP and TCP.

The getnameinfo() function shall be thread-safe.

RETURN VALUE

A zero return value for getnameinfo() indicates successful completion; a non-zero return value indicates failure. The possible values for the failures are listed in the ERRORS section.

Upon successful completion, getnameinfo() shall return the node and service names, if requested, in the buffers provided. The returned names are always null-terminated strings.

ERRORS

The getnameinfo() function shall fail and return the corresponding value if:

[EAI_AGAIN] The name could not be resolved at this time. Future attempts may succeed.

[EAI_BADFLAGS] The flags had an invalid value.

[EAI_FAIL] A non-recoverable error occurred.

[EAI_FAMILY] The address family was not recognized or the address length was invalid for the specified family.

[EAI_MEMORY] There was a memory allocation failure.

[EAI_NONAME] The name does not resolve for the supplied parameters.

NI_NAMEREQD is set and the host's name cannot be located, or both nodename and servname were null.

[EAI_SYSTEM] A system error occurred. The error code can be found in errno.

EXAMPLES

None.

APPLICATION USAGE

If the returned values are to be used as part of any further name resolution (for example, passed to getaddrinfo()), applications should provide buffers large enough to store any result possible on the system.

Given the IPv4-mapped IPv6 address "::ffff:1.2.3.4", the implementation performs a lookup as if the socket address structure contains the IPv4 address "1.2.3.4".

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

gai_strerror(), getaddrinfo(), getservbyname(), inet_ntop(), socket(), the Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>, <sys/socket.h>

CHANGE HISTORY

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

The restrict keyword is added to the getnameinfo() prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
getnetbyaddr, getnetbyname, getnetent — network database functions

SYNOPSIS
#include <netdb.h>

struct netent *getnetbyaddr(uint32_t net, int type);
struct netent *getnetbyname(const char *name);
struct netent *getnetent(void);

DESCRIPTION
Refer to endnetent().
NAME
getopt, optarg, opterr, optind, optopt — command option parsing

SYNOPSIS
#include <unistd.h>

int getopt(int argc, char * const argv[], const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;

DESCRIPTION
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.

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.

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.

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.

If, when getopt() is called:

argv[optind] is a null pointer
*argv[optind] is not the character –
argv[optind] points to the string "--"

getopt() shall return -1 without changing optind. If:

argv[optind] points to the string "--"

getopt() shall return -1 after incrementing optind.

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.

The getopt() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.
getopt()  

**RETURN VALUE**

The `getopt()` function shall return the next option character specified on the command line.

A colon (':') shall be returned if `getopt()` detects a missing argument and the first character of `optstring` was a colon (':').

A question mark ('?') shall be returned if `getopt()` encounters an option character not in `optstring` or detects a missing argument and the first character of `optstring` was not a colon (':').

Otherwise, `getopt()` shall return −1 when all command line options are parsed.

**ERRORS**

No errors are defined.

**EXAMPLES**

**Parsing Command Line Options**

The following code fragment shows how you might process the arguments for a utility that can take the mutually-exclusive options `a` and `b` and the options `f` and `o`, both of which require arguments:

```c
#include <unistd.h>

int main(int argc, char *argv[]) {
    int c;
    int bflg, aflg, errflg;
    char *ifile;
    char *ofile;
    extern char *optarg;
    extern int optind, optopt;
    ...
    while ((c = getopt(argc, argv, "abf:o")) != -1) {
        switch(c) {
            case 'a':
                if (bflg)
                    errflg++;
                else
                    aflg++;
                break;
            case 'b':
                if (aflg)
                    errflg++;
                else {
                    bflg++;
                    bproc();
                }
                break;
            case 'f':
                ifile = optarg;
                break;
            case 'o':
                ofile = optarg;
                break;
        }
    }
    return 0;
}
```

530 System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
case ':':    /* -f or -o without operand */
    fprintf(stderr, 
      "Option \%-c requires an operand\n", optopt);
    errflg++;
    break;
    case '?':
      fprintf(stderr, 
        "Unrecognized option: \%-c\n", optopt);
      errflg++;
      }
      }
      if (errflg) {
        fprintf(stderr, "usage: ... ");
        exit(2);
        }
      for ( ; optind < argc; optind++) {
        if (access(argv[optind], R_OK)) {
          ...
        }
    }
    This code accepts any of the following as equivalent:
    cmd -ao arg path path
    cmd -a -o arg path path
    cmd -o arg -a path path
    cmd -a -o arg -- path path
    cmd -a -o arg path path
    cmd -ao arg path path

    Checking Options and Arguments

    The following example parses a set of command line options and prints messages to standard
    output for each option and argument that it encounters.
    
    #include <unistd.h>
    #include <stdio.h>
    ...
    int c;
    char *filename;
    extern char *optarg;
    extern int optind, optopt, opterr;
    ...
    while ((c = getopt(argc, argv, ":abf:")) != -1) {
      switch(c) {
      case 'a':
        printf("a is set\n");
        break;
      case 'b':
        printf("b is set\n");
        break;
      case 'f':
        filename = optarg;
        printf("filename is %s\n", filename);
        break;
case ':':
    printf("-%c without filename\n", optopt);
    break;

case '?':
    printf("unknown arg %c\n", optopt);
    break;
}

Selecting Options from the Command Line

The following example selects the type of database routines the user wants to use based on the Options argument.

#include <unistd.h>
#include <string.h>

... char *Options = "hdbtl";
... int dbtype, i;
char c;
char *st;
... dbtype = 0;
while ((c = getopt(argc, argv, Options)) != -1) {
    if ((st = strchr(Options, c)) != NULL) {
        dbtype = st - Options;
        break;
    }
}

APPLICATION USAGE

The getopt() function is only required to support option characters included in Utility Syntax Guideline 3. Many historical implementations of getopt() support other characters as options. This is an allowed extension, but applications that use extensions are not maximally portable. Note that support for multi-byte option characters is only possible when such characters can be represented as type int.

RATIONALE

The optopt variable represents historical practice and allows the application to obtain the identity of the invalid option.

The description has been written to make it clear that getopt(), like the getopts utility, deals with option-arguments whether separated from the option by <blank>s or not. Note that the requirements on getopt() and getopts are more stringent than the Utility Syntax Guidelines.

The getopt() function shall return -1, rather than EOF, so that <stdio.h> is not required.

The special significance of a colon as the first character of optstring makes getopt() consistent with the getopts utility. It allows an application to make a distinction between a missing argument and an incorrect option letter without having to examine the option letter. It is true that a missing argument can only be detected in one case, but that is a case that has to be considered.
FUTURE DIRECTIONS
17540 None.

SEE ALSO
17542 exec, the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>, the Shell and Utilities volume of IEEE Std 1003.1-2001

CHANGE HISTORY
17544 First released in Issue 1. Derived from Issue 1 of the SVID.
17546 Issue 5
17547 A note indicating that the getopt() function need not be reentrant is added to the DESCRIPTION.
17548 Issue 6
17549 IEEE PASC Interpretation 1003.2 #150 is applied.
NAME
getpeername — get the name of the peer socket

SYNOPSIS
#include <sys/socket.h>

int getpeername(int socket, struct sockaddr *restrict address,
    socklen_t *restrict address_len);

DESCRIPTION
The getpeername() function shall retrieve the peer address of the specified socket, store this
address in the sockaddr structure pointed to by the address argument, and store the length of this
address in the object pointed to by the address_len argument.

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

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

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to
indicate the error.

ERRORS
The getpeername() function shall fail if:

[EBADF] The socket argument is not a valid file descriptor.
[EINVAL] The socket has been shut down.
[ENOTCONN] The socket is not connected or otherwise has not had the peer pre-specified.
[ENOTSOCK] The socket argument does not refer to a socket.
[EOPNOTSUPP] The operation is not supported for the socket protocol.

The getpeername() function may fail if:

[ENOBUFS] Insufficient resources were available in the system to complete the call.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
accept(), bind(), getsockname(), socket(), the Base Definitions volume of IEEE Std 1003.1-2001,
/sys/socket.h>

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

The restrict keyword is added to the getpeername() prototype for alignment with the
NAME
getpgid — get the process group ID for a process

SYNOPSIS
XSI
#include <unistd.h>

pid_t getpgid(pid_t pid);

DESCRIPTION
The getpgid() function shall return the process group ID of the process whose process ID is equal to \textit{pid}. If \textit{pid} is equal to 0, getpgid() shall return the process group ID of the calling process.

RETURN VALUE
Upon successful completion, getpgid() shall return a process group ID. Otherwise, it shall return (\textit{pid_t})\textasciitilde1 and set \textit{errno} to indicate the error.

ERRORS
The getpgid() function shall fail if:

- [EPERM] The process whose process ID is equal to \textit{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.

- [ESRCH] There is no process with a process ID equal to \textit{pid}.

The getpgid() function may fail if:

- [EINVAL] The value of the \textit{pid} argument is invalid.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
exec, fork(), getpgrp(), getpid(), getsid(), setpgid(), setsid(), the Base Definitions volume of IEEE Std 1003.1-2001, \textit{<unistd.h>}

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
getpgroup()  

NAME  
getpgroup — get the process group ID of the calling process  

SYNOPSIS  
#include <unistd.h>  
pid_t getpgroup(void);  

DESCRIPTION  
The getpgroup() function shall return the process group ID of the calling process.  

RETURN VALUE  
The getpgroup() function shall always be successful and no return value is reserved to indicate an error.  

ERRORS  
No errors are defined.  

EXAMPLES  
None.  

APPLICATION USAGE  
None.  

RATIONALE  
4.3 BSD provides a getpgroup() function that returns the process group ID for a specified process. Although this function supports job control, all known job control shells always specify the calling process with this function. Thus, the simpler System V getpgroup() suffices, and the added complexity of the 4.3 BSD getpgroup() is provided by the XSI extension getpgid().  

FUTURE DIRECTIONS  
None.  

SEE ALSO  
exe, fork(), getpgid(), getpid(), getppid(), kill(), setpgid(), setsid(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>  

CHANGE HISTORY  
First released in Issue 1. Derived from Issue 1 of the SVID.  

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.
NAME
getpid — get the process ID

SYNOPSIS
#include <unistd.h>

pid_t getpid(void);

DESCRIPTION
The getpid() function shall return the process ID of the calling process.

RETURN VALUE
The getpid() function shall always be successful and no return value is reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
evac, fork(), getpgrp(), getppid(), kill(), setpgid(), setsid(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

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.
getpmsg() — receive next message from a STREAMS file

**SYNOPSIS**

```c
#include <stropts.h>

int getpmsg(int fildes, struct strbuf *restrict ctlptr,
              struct strbuf *restrict dataptr, int *restrict bandp,
              int *restrict flagsp);
```

**DESCRIPTION**

Refer to `getmsg()`.
NAME
getppid — get the parent process ID

SYNOPSIS
#include <unistd.h>

pid_t getppid(void);

DESCRIPTION
The getppid() function shall return the parent process ID of the calling process.

RETURN VALUE
The getppid() function shall always be successful and no return value is reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
eexec(), fork(), getpgid(), getpgrp(), getpid(), kill(), setpgid(), setsid(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

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.
**NAME**

getpriority(), setpriority — get and set the nice value

**SYNOPSIS**

```c
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int value);
```

**DESCRIPTION**

The `getpriority()` function shall obtain the nice value of a process, process group, or user. The `setpriority()` function shall set the nice value of a process, process group, or user to `value + {NZERO}`.

Target processes are specified by the values of the `which` and `who` arguments. The `which` argument may be one of the following values: `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, indicating that the `who` argument is to be interpreted as a process ID, a process group ID, or an effective user ID, respectively. A 0 value for the `who` argument specifies the current process, process group, or user.

The nice value set with `setpriority()` shall be applied to the process. If the process is multi-threaded, the nice value shall affect all system scope threads in the process.

If more than one process is specified, `getpriority()` shall return value `−1` less than the lowest nice value pertaining to any of the specified processes, and `setpriority()` shall set the nice values of all of the specified processes to `value + {NZERO}`.

The default nice value is `{NZERO}`; lower nice values shall cause more favorable scheduling.

While the range of valid nice values is `[0, {NZERO}*2−1]`, implementations may enforce more restrictive limits. If `value + {NZERO}` is less than the system’s lowest supported nice value, `setpriority()` shall set the nice value to the lowest supported value; if `value + {NZERO}` is greater than the system’s highest supported nice value, `setpriority()` shall set the nice value to the highest supported value.

Only a process with appropriate privileges can lower its nice value.

Any processes or threads using `SCHED_FIFO` or `SCHED_RR` shall be unaffected by a call to `setpriority()`. This is not considered an error. A process which subsequently reverts to `SCHED_OTHER` need not have its priority affected by such a `setpriority()` call.

The effect of changing the nice value may vary depending on the process-scheduling algorithm in effect.

Since `getpriority()` can return the value `−1` on successful completion, it is necessary to set `errno` to 0 prior to a call to `getpriority()`. If `getpriority()` returns the value `−1`, then `errno` can be checked to see if an error occurred or if the value is a legitimate nice value.

Upon successful completion, `getpriority()` shall return an integer in the range `−{NZERO}` to `{NZERO}−1`. Otherwise, `−1` shall be returned and `errno` set to indicate the error.

Upon successful completion, `setpriority()` shall return 0; otherwise, `−1` shall be returned and `errno` set to indicate the error.

**ERRORS**

The `getpriority()` and `setpriority()` functions shall fail if:

- `[ESRCH]` No process could be located using the `which` and `who` argument values specified.
getpriority()

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.

In addition, setpriority() may fail if:

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.

A request was made to change the nice value to a lower numeric value and the current process does not have appropriate privileges.

EXAMPLES

Using getpriority()

The following example returns the current scheduling priority for the process ID returned by the call to getpid().

```c
#include <sys/resource.h>
...
int which = PRIO_PROCESS;
id_t pid;
int ret;
pid = getpid();
ret = getpriority(which, pid);
```

Using setpriority()

The following example sets the priority for the current process ID to -20.

```c
#include <sys/resource.h>
...
int which = PRIO_PROCESS;
id_t pid;
int priority = -20;
int ret;
pid = getpid();
ret = setpriority(which, pid, priority);
```

APPLICATION USAGE

The getpriority() and setpriority() functions work with an offset nice value (nice value -|NZERO|). The nice value is in the range [0,2*|NZERO| -1], while the return value for getpriority() and the third parameter for setpriority() are in the range [-|NZERO|,|NZERO| -1].

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

nice(), sched_get_priority_max(), sched_setscheduler(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/resource.h>
getpriority()

17820 CHANGE HISTORY
17821 First released in Issue 4, Version 2.
17822 **Issue 5**
17823 Moved from X/OPEN UNIX extension to BASE.
17824 The DESCRIPTION is reworded in terms of the nice value rather than *priority* to avoid confusion with functionality in the POSIX Realtime Extension.
NAME
getprotobynumber, getprotobynumber, getprotoent — network protocol database functions

SYNOPSIS
#include <netdb.h>

struct protoent *getprotobyname(const char *name);
struct protoent *getprotobynumber(int proto);
struct protoent *getprotoent(void);

DESCRIPTION
Refer to endprotoent().
getpwent()

NAME
getpwent — get user database entry

SYNOPSIS
#ifndef __XSI__
#include <pwd.h>
#endif

struct passwd *getpwent(void);

DESCRIPTION
Refer to endpwent().
NAME
getpwnam, getpwnam_r — search user database for a name

SYNOPSIS
#include <pwd.h>

struct passwd *getpwnam(const char *name);

TSF
int getpwnam_r(const char *name, struct passwd *pwd, char *buffer,
size_t bufsize, struct passwd **result);

DESCRIPTION
The getpwnam() function shall search the user database for an entry with a matching name.
The getpwnam() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.
Applications wishing to check for error situations should set errno to 0 before calling getpwnam(). If getpwnam() returns a null pointer and errno is non-zero, an error occurred.

The getpwnam_r() function shall update the passwd structure pointed to by pwd and store a pointer to that structure at the location pointed to by result. The structure shall contain an entry from the user database with a matching name. Storage referenced by the structure is allocated from the memory provided with the buffer parameter, which is bufsize bytes in size. The maximum size needed for this buffer can be determined with the {SC_GETPW_R_SIZE_MAX} sysconf() parameter. A NULL pointer shall be returned at the location pointed to by result on error or if the requested entry is not found.

RETURN VALUE
The getpwnam() function shall return a pointer to a struct passwd with the structure as defined in <pwd.h> with a matching entry if found. A null pointer shall be returned if the requested entry is not found, or an error occurs. On error, errno shall be set to indicate the error.
The return value may point to a static area which is overwritten by a subsequent call to getpwent(), getpwnam(), or getpwuid().

If successful, the getpwnam_r() function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS
The getpwnam() and getpwnam_r() functions may fail if:

[EIO] An I/O error has occurred.

[EINTR] A signal was caught during getpwnam().

[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

[ENFILE] The maximum allowable number of files is currently open in the system.

The getpwnam_r() function may fail if:

[ERANGE] Insufficient storage was supplied via buffer and bufsize to contain the data to be referenced by the resulting passwd structure.
getpwnam()

EXAMPLES

Getting an Entry for the Login Name

The following example uses the \texttt{getlogin()} function to return the name of the user who logged in; this information is passed to the \texttt{getpwnam()} function to get the user database entry for that user.

```c
#include <sys/types.h>
#include <pwd.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

char *lgn;
struct passwd *pw;

if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
    fprintf(stderr, "Get of user information failed.\n"); exit(1);
}
```

APPLICATION USAGE

Three names associated with the current process can be determined: \texttt{getpwuid(geteuid())} returns the name associated with the effective user ID of the process; \texttt{getlogin()} returns the name associated with the current login activity; and \texttt{getpwuid(getuid())} returns the name associated with the real user ID of the process.

The \texttt{getpwnam_r()} function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

\texttt{getpwuid()}, the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<limits.h>}, \texttt{<pwd.h>}, \texttt{<sys/types.h>}

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The \texttt{getpwnam_r()} function is included for alignment with the POSIX Threads Extension.

A note indicating that the \texttt{getpwnam()} function need not be reentrant is added to the DESCRIPTION.

Issue 6

The \texttt{getpwnam_r()} function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION describing matching the \texttt{name}. 
In the SYNOPTIS, the optional include of the `<sys/types.h>` header is removed.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

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 RETURN VALUE section, the requirement to set `errno` on error is added.

- The `[EMFILE]`, `[ENFILE]`, and `[ENXIO]` optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from `bufsize` characters to bytes.
NAME
getpwuid, getpwuid_r — search user database for a user ID

SYNOPSIS
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

TSF
int getpwuid_r(uid_t uoid, struct passwd *pwd, char *buffer, size_t bufsize, struct passwd **result);

DESCRIPTION
The getpwuid() function shall search the user database for an entry with a matching uid.
The getpwuid() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.
Applications wishing to check for error situations should set errno to 0 before calling getpwuid().
If getpwuid() returns a null pointer and errno is set to non-zero, an error occurred.
The getpwuid_r() function shall update the passwd structure pointed to by pwd and store a pointer to that structure at the location pointed to by result. The structure shall contain an entry from the user database with a matching uid. Storage referenced by the structure is allocated from the memory provided with the buffer parameter, which is bufsize bytes in size. The maximum size needed for this buffer can be determined with the [_SC_GETPW_R_SIZE_MAX] sysconf() parameter. A NULL pointer shall be returned at the location pointed to by result on error or if the requested entry is not found.

RETURN VALUE
The getpwuid() function shall return a pointer to a struct passwd with the structure as defined in <pwd.h> with a matching entry if found. A null pointer shall be returned if the requested entry is not found, or an error occurs. On error, errno shall be set to indicate the error.
The return value may point to a static area which is overwritten by a subsequent call to getpwent(), getpwnam(), or getpwuid().
If successful, the getpwuid_r() function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS
The getpwuid() and getpwuid_r() functions may fail if:
[EIO] An I/O error has occurred.
[EINTR] A signal was caught during getpwuid().
[EMFILE] (OPEN_MAX) file descriptors are currently open in the calling process.
[ENFILE] The maximum allowable number of files is currently open in the system.
The getpwuid_r() function may fail if:
[ERANGE] Insufficient storage was supplied via buffer and bufsize to contain the data to be referenced by the resulting passwd structure.
**Examples**

**Getting an Entry for the Root User**

The following example gets the user database entry for the user with user ID 0 (root).

```c
#include <sys/types.h>
#include <pwd.h>
...
uid_t id = 0;
struct passwd *pwd;
pwd = getpwuid(id);
```

**Finding the Name for the Effective User ID**

The following example defines `pws` as a pointer to a structure of type `passwd`, which is used to store the structure pointer returned by the call to the `getpwuid()` function. The `geteuid()` function shall return the effective user ID of the calling process; this is used as the search criteria for the `getpwuid()` function. The call to `getpwuid()` shall return a pointer to the structure containing that user ID value.

```c
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
...
struct passwd *pws;
pws = getpwuid(geteuid());
```

**Finding an Entry in the User Database**

The following example uses `getpwuid()` to search the user database for a user ID that was previously stored in a `stat` structure, then prints out the user name if it is found. If the user is not found, the program prints the numeric value of the user ID for the entry.

```c
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
...
struct stat statbuf;
struct passwd *pwd;
...
if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
  printf(" %-8.8s", pwd->pw_name);
else
  printf(" %-8d", statbuf.st_uid);
```

**Application Usage**

Three names associated with the current process can be determined: `getpwuid(geteuid())` returns the name associated with the effective user ID of the process; `getlogin()` returns the name associated with the current login activity; and `getpwuid(getuid())` returns the name associated with the real user ID of the process.

The `getpwuid_r()` function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.
getpwuid()

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getpwnam(), geteuid(), getuid(), getlogin(), the Base Definitions volume of IEEE Std 1003.1-2001,
<limits.h>, <pwd.h>, <sys/types.h>

CHANGE HISTORY
First released in Issue 1. Derived from System V Release 2.0.

Issue 5
Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.
The getpwuid_r() function is included for alignment with the POSIX Threads Extension.
A note indicating that the getpwuid() function need not be reentrant is added to the DESCRIPTION.

Issue 6
The getpwuid_r() function is marked as part of the Thread-Safe Functions option.
The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION describing matching the uid.
In the SYNOPSIS, the optional include of the <sys/types.h> header is removed.
In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
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 RETURN VALUE section, the requirement to set errno on error is added.
• The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.
The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.
IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from bufsize characters to bytes.
NAME
getrlimit, setrlimit — control maximum resource consumption

SYNOPSIS
XSI
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, const struct rlimit *rlp);

DESCRIPTION
The getrlimit() function shall get, and the setrlimit() function shall set, limits on the consumption
of a variety of resources.

Each call to either getrlimit() or setrlimit() identifies a specific resource to be operated upon as
well as a resource limit. A resource limit is represented by an rlimit structure. The rlim_cur
member specifies the current or soft limit and the rlim_max member specifies the maximum or
hard limit. Soft limits may be changed by a process to any value that is less than or equal to the
hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or
equal to the soft limit. Only a process with appropriate privileges can raise a hard limit. Both
hard and soft limits can be changed in a single call to setrlimit() subject to the constraints
described above.

The value RLIM_INFINITY, defined in <sys/resource.h>, shall be considered to be larger than
any other limit value. If a call to getrlimit() returns RLIM_INFINITY for a resource, it means the
implementation shall not enforce limits on that resource. Specifying RLIM_INFINITY as any
resource limit value on a successful call to setrlimit() shall inhibit enforcement of that resource
limit.

The following resources are defined:

RLIMIT_CORE This is the maximum size of a core file, in bytes, that may be created by a
process. A limit of 0 shall prevent the creation of a core file. If this limit is
exceeded, the writing of a core file shall terminate at this size.

RLIMIT_CPU This is the maximum amount of CPU time, in seconds, used by a process.
If this limit is exceeded, SIGXCPU shall be generated for the process. If
the process is catching or ignoring SIGXCPU, or all threads belonging to
that process are blocking SIGXCPU, the behavior is unspecified.

RLIMIT_DATA This is the maximum size of a process’ data segment, in bytes. If this limit
is exceeded, the malloc() function shall fail with errno set to [ENOMEM].

RLIMIT_FSIZE This is the maximum size of a file, in bytes, that may be created by a
process. If a write or truncate operation would cause this limit to be
exceeded, SIGXFSZ shall be generated for the thread. If the thread is
blocking, or the process is catching or ignoring SIGXFSZ, continued
attempts to increase the size of a file from end-of-file to beyond the limit
shall fail with errno set to [EFBIG].

RLIMIT_NOFILE This is a number one greater than the maximum value that the system
may assign to a newly-created descriptor. If this limit is exceeded,
functions that allocate new file descriptors may fail with errno set to
[EMFILE]. This limit constrains the number of file descriptors that a
process may allocate.

RLIMIT_STACK This is the maximum size of a process’ stack, in bytes. The
implementation does not automatically grow the stack beyond this limit.
If this limit is exceeded, SIGSEGV shall be generated for the thread. If the thread is blocking SIGSEGV, or the process is ignoring or catching SIGSEGV and has not made arrangements to use an alternate stack, the disposition of SIGSEGV shall be set to SIG_DFL before it is generated.

**RLIMIT_AS** This is the maximum size of a process' total available memory, in bytes. If this limit is exceeded, the `malloc()` and `mmap()` functions shall fail with *errno* set to [ENOMEM]. In addition, the automatic stack growth fails with the effects outlined above.

When using the `getrlimit()` function, if a resource limit can be represented correctly in an object of type `rlim_t`, then its representation is returned; otherwise, if the value of the resource limit is equal to that of the corresponding saved hard limit, the value returned shall be RLIM_SAVED_MAX; otherwise, the value returned shall be RLIM_SAVED_CUR.

When using the `setrlimit()` function, if the requested new limit is RLIM_INFINITY, the new limit shall be “no limit”; otherwise, if the requested new limit is RLIM_SAVED_MAX, the new limit shall be the corresponding saved hard limit; otherwise, if the requested new limit is RLIM_SAVED_CUR, the new limit shall be the corresponding saved soft limit; otherwise, the new limit shall be the requested value. In addition, if the corresponding saved limit can be represented correctly in an object of type `rlim_t` then it shall be overwritten with the new limit.

The result of setting a limit to RLIM_SAVED_MAX or RLIM_SAVED_CUR is unspecified unless a previous call to `getrlimit()` returned that value as the soft or hard limit for the corresponding resource limit.

The determination of whether a limit can be correctly represented in an object of type `rlim_t` is implementation-defined. For example, some implementations permit a limit whose value is greater than RLIM_INFINITY and others do not.

The `exec` family of functions shall cause resource limits to be saved.

**RETURN VALUE**

Upon successful completion, `getrlimit()` and `setrlimit()` shall return 0. Otherwise, these functions shall return −1 and set *errno* to indicate the error.

**ERRORS**

The `getrlimit()` and `setrlimit()` functions shall fail if:

- **[EINVAL]** An invalid resource was specified; or in a `setrlimit()` call, the new `rlim_cur` exceeds the new `rlim_max`.

- **[EPERM]** The limit specified to `setrlimit()` would have raised the maximum limit value, and the calling process does not have appropriate privileges.

The `setrlimit()` function may fail if:

- **[EINVAL]** The limit specified cannot be lowered because current usage is already higher than the limit.
getrlimit()

EXAMPLES
None.

APPLICATION USAGE
If a process attempts to set the hard limit or soft limit for RLIMIT_NOFILE to less than the value of \_POSIX_OPEN_MAX from <limits.h>, unexpected behavior may occur.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
exec, fork(), malloc(), open(), sigaltstack(), sysconf(), ulimit(), the Base Definitions volume of IEEE Std 1003.1-2001, <stropts.h>, <sys/resource.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
An APPLICATION USAGE section is added.
Large File Summit extensions are added.
getrusage()

NAME
getrusage — get information about resource utilization

SYNOPSIS
#include <sys/resource.h>
int getrusage(int who, struct rusage *r_usage);

DESCRIPTION
The getrusage() function shall provide measures of the resources used by the current process or its terminated and waited-for child processes. If the value of the who argument is RUSAGE_SELF, information shall be returned about resources used by the current process. If the value of the who argument is RUSAGE_CHILDREN, information shall be returned about resources used by the terminated and waited-for children of the current process. If the child is never waited for (for example, if the parent has SA_NOCLDWAIT set or sets SIGCHLD to SIG_IGN), the resource information for the child process is discarded and not included in the resource information provided by getrusage().

The r_usage argument is a pointer to an object of type struct rusage in which the returned information is stored.

RETURN VALUE
Upon successful completion, getrusage() shall return 0; otherwise, −1 shall be returned and errno set to indicate the error.

ERRORS
The getrusage() function shall fail if:

[EINVAL] The value of the who argument is not valid.

EXAMPLES
Using getrusage()
The following example returns information about the resources used by the current process.

#include <sys/resource.h>
...
int who = RUSAGE_SELF;
struct rusage usage;
int ret;
ret = getrusage(who, &usage);

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
exit(), sigaction(), time(), times(), wait(), the Base Definitions volume of IEEE Std 1003.1-2001,
<sys/resource.h>
18192 **CHANGE HISTORY**
18193 First released in Issue 4, Version 2.
18194 **Issue 5**
18195 Moved from X/OPEN UNIX extension to BASE.
gets() System Interfaces

NAME
gets — get a string from a stdin stream

SYNOPSIS
#include <stdio.h>
char *gets(char *s);

DESCRIPTION
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 gets() function shall read bytes from the standard input stream, stdin, into the array pointed to by s, until a <newline> is read or an end-of-file condition is encountered. Any <newline> shall be discarded and a null byte shall be placed immediately after the last byte read into the array.

The gets() 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(), fread(), getc(), getchar(), gets(), fscanf(), or scanf() using stream that returns data not supplied by a prior call to ungetc().

RETURN VALUE
Upon successful completion, gets() shall return s. If the stream is at end-of-file, the end-of-file indicator for the stream shall be set and gets() shall return a null pointer. If a read error occurs, the error indicator for the stream shall be set, gets() shall return a null pointer, and set errno to indicate the error.

ERRORS
Refer to fgetc().

APPLICATION USAGE
Reading a line that overflows the array pointed to by s results in undefined behavior. The use of fgets() is recommended.

Since the user cannot specify the length of the buffer passed to gets(), use of this function is discouraged. The length of the string read is unlimited. It is possible to overflow this buffer in such a way as to cause applications to fail, or possible system security violations.

It is recommended that the fgets() function should be used to read input lines.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
feof(), ferror(), fgets(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.
NAME
getservbyname, getservbyport, getservent — network services database functions

SYNOPSIS
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);
struct servent *getservent(void);

DESCRIPTION
Refer to endservent().
NAME
getsid — get the process group ID of a session leader

SYNOPSIS
XSI
#include <unistd.h>

pid_t getsid(pid_t pid);

DESCRIPTION
The getsid() function shall obtain the process group ID of the process that is the session leader of
the process specified by pid. If pid is (pid_t)0, it specifies the calling process.

RETURN VALUE
Upon successful completion, getsid() shall return the process group ID of the session leader of
the specified process. Otherwise, it shall return (pid_t)−1 and set errno to indicate the error.

ERRORS
The getsid() function shall fail if:

[EPERM] The process specified by pid is not in the same session as the calling process,
and the implementation does not allow access to the process group ID of the
session leader of that process from the calling process.

[ESRCH] There is no process with a process ID equal to pid.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
eexec, fork(), getpid(), getpgid(), setpgid(), setsid(), the Base Definitions volume of
IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
NAME
getsockname — get the socket name

SYNOPSIS
#include <sys/socket.h>
int getsockname(int socket, struct sockaddr *restrict address,
socklen_t *restrict address_len);

DESCRIPTION
The getsockname() function shall retrieve the locally-bound name of the specified socket, store
this address in the sockaddr structure pointed to by the address argument, and store the length of
this address in the object pointed to by the address_len argument.
If the actual length of the address is greater than the length of the supplied sockaddr structure,
the stored address shall be truncated.
If the socket has not been bound to a local name, the value stored in the object pointed to by
address is unspecified.

RETURN VALUE
Upon successful completion, 0 shall be returned, the address argument shall point to the address
of the socket, and the address_len argument shall point to the length of the address. Otherwise, −1
shall be returned and errno set to indicate the error.

ERRORS
The getsockname() function shall fail if:
[EBADF] The socket argument is not a valid file descriptor.
[ENOTSOCK] The socket argument does not refer to a socket.
[EOPNOTSUPP] The operation is not supported for this socket’s protocol.
The getsockname() function may fail if:
[EINVAL] The socket has been shut down.
[ENOBUFS] Insufficient resources were available in the system to complete the function.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
accept(), bind(), getpeername(), socket(), the Base Definitions volume of IEEE Std 1003.1-2001,
<sys/socket.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
The restrict keyword is added to the getsockname() prototype for alignment with the
NAME
getsockopt — get the socket options

SYNOPSIS
#include <sys/socket.h>

int getsockopt(int socket, int level, int option_name,
    void *restrict option_value, socklen_t *restrict option_len);

DESCRIPTION
The getsockopt() function manipulates options associated with a socket.

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.

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.

The socket in use may require the process to have appropriate privileges to use the getsockopt() function.

The option_name argument specifies a single option to be retrieved. It can be one of the following values defined in <sys/socket.h>:

SO_DEBUG Reports whether debugging information is being recorded. This option shall store an int value. This is a Boolean option.

SO_ACCEPTCONN Reports whether socket listening is enabled. This option shall store an int value. This is a Boolean option.

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.

SO_REUSEADDR Reports whether the rules used in validating addresses supplied to bind() 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.

SO_KEEPALIVE Reports whether connections are kept active with periodic transmission of messages, if this is supported by the protocol.

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.

SO_LINGER Reports whether the socket lingers on close() if data is present. If SO_LINGER is set, the system blocks the process during close() until it can transmit the data or until the end of the interval indicated by the l linger member, whichever comes first. If SO_LINGER is not specified, and close() is issued, the system handles the call in a way that allows the process to continue as quickly as possible. This option shall store a linger structure.
getsockopt()

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.

SO_SNDBUF Reports send buffer size information. This option shall store an int value.

SO_RCVBUF Reports receive buffer size information. This option shall store an int value.

SO_ERROR Reports information about error status and clears it. This option shall store an int value.

SO_TYPE Reports the socket type. This option shall store an int value. Socket types are described in Section 2.10.6 (on page 59).

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.

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.

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 errno 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.

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.

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 errno 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.

For Boolean options, a zero value indicates that the option is disabled and a non-zero value indicates that the option is enabled.
Options at other protocol levels vary in format and name.

The socket in use may require the process to have appropriate privileges to use the `getsockopt()` function.

**RETURN VALUE**
Upon successful completion, `getsockopt()` shall return 0; otherwise, −1 shall be returned and `errno` set to indicate the error.

**ERRORS**
The `getsockopt()` function shall fail if:

- **[EBADF]** The `socket` argument is not a valid file descriptor.
- **[EINVAL]** The specified option is invalid at the specified socket level.
- **[ENOPROTOOPT]** The option is not supported by the protocol.
- **[ENOTSOCK]** The `socket` argument does not refer to a socket.

The `getsockopt()` function may fail if:

- **[EACCES]** The calling process does not have the appropriate privileges.
- **[EINVAL]** The socket has been shut down.
- **[ENOBUFS]** Insufficient resources are available in the system to complete the function.

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`bind()`, `close()`, `endprotoent()`, `setsockopt()`, `socket()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/socket.h>`, `<netinet/in.h>`

**CHANGE HISTORY**
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The `restrict` keyword is added to the `getsockopt()` prototype for alignment with the ISO/IEC 9899:1999 standard.
**NAME**
getsubopt — parse suboption arguments from a string

**SYNOPSIS**

```c
#include <stdlib.h>

int getsubopt(char **optionp, char * const *tokens, char **valuep);
```

**DESCRIPTION**

The `getsubopt()` function shall parse suboption arguments in a flag argument. Such options often result from the use of `getopt()`.

The `getsubopt()` argument `optionp` is a pointer to a pointer to the option argument string. The suboption arguments shall be separated by commas and each may consist of either a single token, or a token-value pair separated by an equal sign.

The `keylistp` argument shall be a pointer to a vector of strings. The end of the vector is identified by a null pointer. Each entry in the vector is one of the possible tokens that might be found in `*optionp`. Since commas delimit suboption arguments in `optionp`, they should not appear in any of the strings pointed to by `keylistp`. Similarly, because an equal sign separates a token from its value, the application should not include an equal sign in any of the strings pointed to by `keylistp`.

The `valuep` argument is the address of a value string pointer.

If a comma appears in `optionp`, it shall be interpreted as a suboption separator. After commas have been processed, if there are one or more equal signs in a suboption string, the first equal sign in any suboption string shall be interpreted as a separator between a token and a value. Subsequent equal signs in a suboption string shall be interpreted as part of the value.

If the string at `*optionp` contains only one suboption argument (equivalently, no commas), `getsubopt()` shall update `*optionp` to point to the null character at the end of the string. Otherwise, it shall isolate the suboption argument by replacing the comma separator with a null character, and shall update `*optionp` to point to the start of the next suboption argument. If the suboption argument has an associated value (equivalently, contains an equal sign), `getsubopt()` shall update `*valuep` to point to the value's first character. Otherwise, it shall set `*valuep` to a null pointer. The calling application may use this information to determine whether the presence or absence of a value for the suboption is an error.

Additionally, when `getsubopt()` fails to match the suboption argument with a token in the `keylistp` array, the calling application should decide if this is an error, or if the unrecognized option should be processed in another way.

**RETURN VALUE**

The `getsubopt()` function shall return the index of the matched token string, or −1 if no token strings were matched.

**ERRORS**

No errors are defined.
EXAMPLES

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

int do_all;
const char *type;
int read_size;
int write_size;
int read_only;

enum
{
    RO_OPTION = 0,
    RW_OPTION,
    READ_SIZE_OPTION,
    WRITE_SIZE_OPTION
};

const char *mount_opts[] =
{
    [RO_OPTION] = "ro",
    [RW_OPTION] = "rw",
    [READ_SIZE_OPTION] = "rsize",
    [WRITE_SIZE_OPTION] = "wsize",
    NULL
};

const char *subopts, *value;
int opt;

while ((opt = getopt(argc, argv, "at:o:")) != -1)
    switch(opt)
    {
    case 'a':
        do_all = 1;
        break;
    case 't':
        type = optarg;
        break;
    case 'o':
        subopts = optarg;
        while (*subopts != '\0')
        {
            switch(getsubopt(&subopts, mount_opts, &value))
            {
            case RO_OPTION:
                read_only = 1;
                break;
            case RW_OPTION:
                read_only = 0;
                break;
            case READ_SIZE_OPTION:
```

```c
```
```
if (value == NULL)
    abort();
read_size = atoi(value);
break;
case WRITE_SIZE_OPTION:
    if (value == NULL)
        abort();
    write_size = atoi(value);
    break;
default: /* Unknown suboption. */
    printf("Unknown suboption '\%s'\n", value);
    break;
    }
    break;
default:
    abort();
}
/* Do the real work. */
return 0;
}

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().

#include <stdlib.h>
...
char *tokens[] = {"HOME", "PATH", "LOGNAME", (char *) NULL };
char *value;
int opt, index;
while ((opt = getopt(argc, argv, "e:")) != -1) {
    switch(opt) {
    case 'e':
        while ((index = getsubopt(&optarg, tokens, &value)) != -1) {
            switch(index) {
                ...
            }
            break;
        }
    ...
    break;
    }
}

APPLICATION USAGE
None.

RATIONALE
None.
getsubopt()

FUTURE DIRECTIONS

None.

SEE ALSO

getopt(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.
NAME
gettimeofday — get the date and time

SYNOPSIS
XSI
#include <sys/time.h>

int gettimeofday(struct timeval *restrict tp, void *restrict tzp);

DESCRIPTION
The gettimeofday() function shall obtain the current time, expressed as seconds and
microseconds since the Epoch, and store it in the timeval structure pointed to by tp. The
resolution of the system clock is unspecified.

If tzp is not a null pointer, the behavior is unspecified.

RETURN VALUE
The gettimeofday() function shall return 0 and no value shall be reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
c_time(),ftime(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/time.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Moved from X/OPEN UNIX extension to BASE.

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.

The restrict keyword is added to the gettimeofday() prototype for alignment with the
NAME
getuid — get a real user ID

SYNOPSIS
#include <unistd.h>
uid_t getuid(void);

DESCRIPTION
The getuid() function shall return the real user ID of the calling process.

RETURN VALUE
The getuid() function shall always be successful and no return value is reserved to indicate the error.

ERRORS
No errors are defined.

EXAMPLES
Setting the Effective User ID to the Real User ID
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.
#include <unistd.h>
#include <sys/types.h>
...
setreuid(getuid(), getuid());
...

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getegid(), geteuid(), getgid(), setegid(), seteuid(), setgid(), setreuid(), setuid(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

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.
getutxent( )

18660 NAME
18661 getutxent, getutcid, getutxline — get user accounting database entries

18662 SYNOPSIS
18663 XSI
18664 #include <utmpx.h>
18665
18666 struct utmpx *getutxent(void);
18667 struct utmpx *getutcid(const struct utmpx *id);
18668 struct utmpx *getutxline(const struct utmpx *line);

18669 DESCRIPTION
18670 Refer to endutxent().
getwc()  

NAME
getwc — get a wide character from a stream

SYNOPSIS
#include <stdio.h>
#include <wchar.h>

wint_t getwc(FILE *stream);

DESCRIPTION
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

The getwc() function shall be equivalent to fgetwc(), except that if it is implemented as a macro it
may evaluate stream more than once, so the argument should never be an expression with side
effects.

RETURN VALUE
Refer to fgetwc().

ERRORS
Refer to fgetwc().

EXAMPLES
None.

APPLICATION USAGE
Since it may be implemented as a macro, getwc() may treat incorrectly a stream argument with
side effects. In particular, getwc(*f++) does not necessarily work as expected. Therefore, use of
this function is not recommended; fgetwc() should be used instead.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fgetwc(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>, <wchar.h>

CHANGE HISTORY
First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
draft.

Issue 5
The Optional Header (OH) marking is removed from <stdio.h>. 

NAME
getwchar — get a wide character from a stdin stream

SYNOPSIS
#include <wchar.h>
wint_t getwchar(void);

DESCRIPTION
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
The getwchar() function shall be equivalent to getwc(stdin).

RETURN VALUE
Refer to fgetwc().

ERRORS
Refer to fgetwc().

EXAMPLES
None.

APPLICATION USAGE
If the wint_t value returned by getwchar() is stored into a variable of type wchar_t and then
compared against the wint_t macro WEOF, the result may be incorrect. Only the wint_t type is
guaranteed to be able to represent any wide character and WEOF.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fgetwc(), getwc(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working
draft.
NAME
getwd — get the current working directory pathname (LEGACY)

SYNOPSIS
XSI
#include <unistd.h>

char *getwd(char *path_name);

DESCRIPTION
The getwd() function shall determine an absolute pathname of the current working directory of
the calling process, and copy a string containing that pathname into the array pointed to by the
path_name argument.

If the length of the pathname of the current working directory is greater than (\{PATH_MAX\}+1)
including the null byte, getwd() shall fail and return a null pointer.

RETURN VALUE
Upon successful completion, a pointer to the string containing the absolute pathname of the
current working directory shall be returned. Otherwise, getwd() shall return a null pointer and
the contents of the array pointed to by path_name are undefined.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
For applications portability, the getcwd() function should be used to determine the current
working directory instead of getwd().

RATIONALE
Since the user cannot specify the length of the buffer passed to getwd(), use of this function is
discouraged. The length of a pathname described in \{PATH_MAX\} is file system-dependent and
may vary from one mount point to another, or might even be unlimited. It is possible to
overflow this buffer in such a way as to cause applications to fail, or possible system security
violations.

It is recommended that the getcwd() function should be used to determine the current working
directory.

FUTURE DIRECTIONS
This function may be withdrawn in a future version.

SEE ALSO
getcwd(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
This function is marked LEGACY.
NAME

glob, globfree — generate pathnames matching a pattern

SYNOPSIS

#include <glob.h>

int glob(const char *restrict pattern, int flags,
    int (*errfunc)(const char *epath, int eerrno),
    glob_t *restrict pglob);

void globfree(glob_t *pglob);

DESCRIPTION

The glob() function is a pathname generator that shall implement the rules defined in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.13, Pattern Matching Notation, with optional support for rule 3 in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.13.3, Patterns Used for Filename Expansion.

The structure type glob_t is defined in <glob.h> and includes at least the following members:

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t</td>
<td>gl_pathc</td>
<td>Count of paths matched by pattern.</td>
</tr>
<tr>
<td>char **</td>
<td>gl_pathv</td>
<td>Pointer to a list of matched pathnames.</td>
</tr>
<tr>
<td>size_t</td>
<td>gl_offs</td>
<td>Slots to reserve at the beginning of gl_pathv.</td>
</tr>
</tbody>
</table>

The argument pattern is a pointer to a pathname pattern to be expanded. The glob() function shall match all accessible pathnames against this pattern and develop a list of all pathnames that match. In order to have access to a pathname, glob() requires search permission on every component of a path except the last, and read permission on each directory of any filename component of pattern that contains any of the following special characters: '*', '?', and '['.

The glob() function shall store the number of matched pathnames into pglob->gl_pathc and a pointer to a list of pointers to pathnames into pglob->gl_pathv. The pathnames shall be in sort order as defined by the current setting of the LC_COLLATE category; see the Base Definitions volume of IEEE Std 1003.1-2001, Section 7.3.2, LC_COLLATE. The first pointer after the last pathname shall be a null pointer. If the pattern does not match any pathnames, the returned number of matched paths is set to 0, and the contents of pglob->gl_pathv are implementation-defined.

It is the caller’s responsibility to create the structure pointed to by pglob. The glob() function shall allocate other space as needed, including the memory pointed to by gl_pathv. The globfree() function shall free any space associated with pglob from a previous call to glob().

The flags argument is used to control the behavior of glob(). The value of flags is a bitwise-inclusive OR of zero or more of the following constants, which are defined in <glob.h>:

- **GLOB_APPEND** Append pathnames generated to the ones from a previous call to glob().
- **GLOB_DOOFFS** Make use of pglob->gl_offs. If this flag is set, pglob->gl_offs is used to specify how many null pointers to add to the beginning of pglob->gl_pathv. In other words, pglob->gl_pathv shall point to pglob->gl_offs null pointers, followed by pglob->gl_pathc pathname pointers, followed by a null pointer.
- **GLOB_ERR** Cause glob() to return when it encounters a directory that it cannot open or read. Ordinarily, glob() continues to find matches.
GLOB_MARK Each pathname that is a directory that matches pattern shall have a slash appended.

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 pattern does not match any pathname, then glob() shall return a list consisting of only pattern, and the number of matched pathnames is 1.

GLOB_NOESCAPE Disable backslash escaping.

GLOB_NOSORT Ordinarily, glob() sorts the matching pathnames according to the current setting of the LC_COLLATE category; see the Base Definitions volume of IEEE Std 1003.1-2001, Section 7.3.2, LC_COLLATE. When this flag is used, the order of pathnames returned is unspecified.

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 errmsg is not a null pointer, glob() calls (*errmsg()) with two arguments:

1. The epath argument is a pointer to the path that failed.
2. The eerrno 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 (*errmsg()) 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 errmsg is a null pointer or (*errmsg()) returns 0, the error shall be ignored.

The glob() function shall not fail because of large files.

**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.
The `globfree()` function shall not return a value.

If `glob()` terminates due to an error, it shall return one of the non-zero constants defined in `<glob.h>`. The arguments `pglob->gl_pathc` and `pglob->gl_pathv` are still set as defined above.

**ERRORS**

The `glob()` function shall fail and return the corresponding value if:

- **GLOB_ABORTED** The scan was stopped because `GLOB_ERR` was set or (*errfunc()) returned non-zero.
- **GLOB_NOMATCH** The pattern does not match any existing pathname, and `GLOB_NOCHECK` was not set in flags.
- **GLOB_NOSPACE** An attempt to allocate memory failed.

**EXAMPLES**

One use of the `GLOB_DOOFFS` flag is by applications that build an argument list for use with `execv()`, `execve()`, or `execvp()`. Suppose, for example, that an application wants to do the equivalent of:

```
ls -l *.c
```

but for some reason:

```
system("ls -l *.c")
```

is not acceptable. The application could obtain approximately the same result using the sequence:

```
globbuf.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &globbuf);
globbuf.gl_pathv[0] = "ls";
globbuf.gl_pathv[1] = "-l";
execvp("ls", &globbuf.gl_pathv[0]);
```

Using the same example:

```
ls -l *.c *.h
```

could be approximately simulated using `GLOB_APPEND` as follows:

```
globbuf.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &globbuf);
glob("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);
...
```

**APPLICATION USAGE**

This function is not provided for the purpose of enabling utilities to perform pathname expansion on their arguments, as this operation is performed by the shell, and utilities are explicitly not expected to redo this. Instead, it is provided for applications that need to do pathname expansion on strings obtained from other sources, such as a pattern typed by a user or read from a file.

If a utility needs to see if a pathname matches a given pattern, it can use `fnmatch()`.

Note that `gl_pathc` and `gl_pathv` have meaning even if `glob()` fails. This allows `glob()` to report partial results in the event of an error. However, if `gl_pathc` is 0, `gl_pathv` is unspecified even if `glob()` did not return an error.

The `GLOB_NOCHECK` option could be used when an application wants to expand a pathname if wildcards are specified, but wants to treat the pattern as just a string otherwise. The `sh` utility
might use this for option-arguments, for example.

The new pathnames generated by a subsequent call with GLOB_APPEND are not sorted together with the previous pathnames. This mirrors the way that the shell handles pathname expansion when multiple expansions are done on a command line.

Applications that need tilde and parameter expansion should use wordexp().

**RATIONALE**

It was claimed that the GLOB_DOOFFS flag is unnecessary because it could be simulated using:

```c
new = (char **)malloc((n + pglob->gl_pathc + 1) * sizeof(char *));
(void) memcpy(new+n, pglob->gl_pathv, pglob->gl_pathc * sizeof(char *));
(void) memset(new, 0, n * sizeof(char *));
free(pglob->gl_pathv);
pglob->gl_pathv = new;
```

However, this assumes that the memory pointed to by gl_pathv is a block that was separately created using malloc(). This is not necessarily the case. An application should make no assumptions about how the memory referenced by fields in pglob was allocated. It might have been obtained from malloc() in a large chunk and then carved up within glob(), or it might have been created using a different memory allocator. It is not the intent of the standard developers to specify or imply how the memory used by glob() is managed.

The GLOB_APPEND flag would be used when an application wants to expand several different patterns into a single list.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

exec, fnmatch(), opendir(), readdir(), stat(), wordexp(), the Base Definitions volume of IEEE Std 1003.1-2001, <glob.h>, the Shell and Utilities volume of IEEE Std 1003.1-2001

**CHANGE HISTORY**


Moved from POSIX2 C-language Binding to BASE.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The restrict keyword is added to the glob() prototype for alignment with the ISO/IEC 9899: 1999 standard.
NAME
gmtime, gmtime_r — convert a time value to a broken-down UTC time

SYNOPSIS
#include <time.h>

struct tm *gmtime(const time_t *timer);

TSF
struct tm *gmtime_r(const time_t *restrict timer,
struct tm *restrict result);

DESCRIPTION
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.

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).

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.

TSF The same relationship shall apply for gmtime_r().

CX The gmtime() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

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.

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.

RETURN VALUE
The gmtime() function shall return a pointer to a struct tm.

TSF Upon successful completion, gmtime_r() shall return the address of the structure pointed to by the argument result. If an error is detected, or UTC is not available, gmtime_r() shall return a null pointer.

ERRORS
No errors are defined.
gmtime()

EXAMPLES
None.

APPLICATION USAGE
The gmtime_r() function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
asctime(), clock(), ctime(), difftime(), localtime(), mktime(), strftime(), strptime(), time(), utime(), the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
A note indicating that the gmtime() function need not be reentrant is added to the DESCRIPTION.

Issue 6
The gmtime_r() function is included for alignment with the POSIX Threads Extension.

Issue 6
The gmtime_r() function is marked as part of the Thread-Safe Functions option.

Extensions beyond the ISO C standard are marked.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The restrict keyword is added to the gmtime_r() prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
grantpt — grant access to the slave pseudo-terminal device

SYNOPSIS
#include <stdlib.h>
int grantpt(int fildes);

DESCRIPTION
The grantpt() function shall change the mode and ownership of the slave pseudo-terminal device associated with its master pseudo-terminal counterpart. The fildes argument is a file descriptor that refers to a master pseudo-terminal device. The user ID of the slave shall be set to the real UID of the calling process and the group ID shall be set to an unspecified group ID. The permission mode of the slave pseudo-terminal shall be set to readable and writable by the owner, and writable by the group.

The behavior of the grantpt() function is unspecified if the application has installed a signal handler to catch SIGCHLD signals.

RETURN VALUE
Upon successful completion, grantpt() shall return 0; otherwise, it shall return -1 and set errno to indicate the error.

ERRORS
The grantpt() function may fail if:

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

EINVAL The fildes argument is not associated with a master pseudo-terminal device.

EACCES The corresponding slave pseudo-terminal device could not be accessed.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
open (), ptsname(), unlockpt (), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.
NAME
h_errno — error return value for network database operations

SYNOPSIS
#include <netdb.h>

DESCRIPTION
This method of returning errors is used only in connection with obsolescent functions.
The <netdb.h> header provides a declaration of h_errno as a modifiable lvalue of type int.
It is unspecified whether h_errno is a macro or an identifier declared with external linkage. If a
macro definition is suppressed in order to access an actual object, or a program defines an
identifier with the name h_errno, the behavior is undefined.

RETURN VALUE
None.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
Applications should obtain the definition of h_errno by the inclusion of the <netdb.h> header.

RATIONALE
None.

FUTURE DIRECTIONS
h_errno may be withdrawn in a future version.

SEE ALSO
endhostent(), errno, the Base Definitions volume of IEEE Std 1003.1-2001, <netdb.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
hcreate, hdestroy, hsearch — manage hash search table

SYNOPSIS
#include <search.h>

int hcreate(size_t nel);
void hdestroy(void);
ENTRY *hsearch(ENTRY item, ACTION action);

DESCRIPTION
The hcreate(), hdestroy(), and hsearch() functions shall manage hash search tables.

The hcreate() function shall allocate sufficient space for the table, and the application shall
ensure it is called before hsearch() is used. The nel argument is an estimate of the maximum
number of entries that the table shall contain. This number may be adjusted upward by the
algorithm in order to obtain certain mathematically favorable circumstances.

The hdestroy() function shall dispose of the search table, and may be followed by another call to
hcreate(). After the call to hdestroy(), the data can no longer be considered accessible.

The hsearch() function is a hash-table search routine. It shall return a pointer into a hash table
indicating the location at which an entry can be found. The item argument is a structure of type
ENTRY (defined in the <search.h> header) containing two pointers: item.key points to the
comparison key (a char *), and item.data (a void *) points to any other data to be associated with
that key. The comparison function used by hsearch() is strcmp(). The action argument is a
member of an enumeration type ACTION indicating the disposition of the entry if it cannot be
found in the table. ENTER indicates that the item should be inserted in the table at an
appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is
indicated by the return of a null pointer.

These functions need not be reentrant. A function that is not required to be reentrant is not
required to be thread-safe.

RETURN VALUE
The hcreate() function shall return 0 if it cannot allocate sufficient space for the table; otherwise,
it shall return non-zero.

The hdestroy() function shall not return a value.

The hsearch() function shall return a null pointer if either the action is FIND and the item could
not be found or the action is ENTER and the table is full.

ERRORS
The hcreate() and hsearch() functions may fail if:

[ENOMEM] Insufficient storage space is available.

EXAMPLES
The following example reads in strings followed by two numbers and stores them in a hash
table, discarding duplicates. It then reads in strings and finds the matching entry in the hash
table and prints it out.

```c
#include <stdio.h>
#include <search.h>
#include <string.h>

struct info { /* This is the info stored in the table */
  int age, room; /* other than the key. */
```
```c
hcreate( )

#define NUM_EMPL 5000 /* # of elements in search table. */

int main(void)
{
    char string_space[NUM_EMPL*20]; /* Space to store strings. */
    struct info info_space[NUM_EMPL]; /* Space to store employee info. */
    char *str_ptr = string_space; /* Next space in string_space. */
    struct info *info_ptr = info_space;
        /* Next space in info_space. */
    ENTRY item;
    ENTRY *found_item; /* Name to look for in table. */
    char name_to_find[30];

    int i = 0;
    /* Create table; no error checking is performed. */
    (void) hcreate(NUM_EMPL);
    while (scanf("%s%d%d", str_ptr, &info_ptr->age,
                 &info_ptr->room) != EOF && i++ < NUM_EMPL) {
        /* Put information in structure, and structure in item. */
        item.key = str_ptr;
        item.data = info_ptr;
        str_ptr += strlen(str_ptr) + 1;
        info_ptr++;
        /* Put item into table. */
        (void) hsearch(item, ENTER);
    }
    /* Access table. */
    item.key = name_to_find;
    while (scanf("%s", item.key) != EOF) {
        if ((found_item = hsearch(item, FIND)) != NULL) {
            /* If item is in the table. */
            (void)printf("found %s, age = %d, room = %d\n",
                         found_item->key,
                         ((struct info *)found_item->data)->age,
                         ((struct info *)found_item->data)->room);
        } else
            (void)printf("no such employee %s\n", name_to_find);
        return 0;
    }
}

APPLICATION USAGE
The hcreate() and hsearch() functions may use malloc() to allocate space.

RATIONALE
None.
```
FUTURE DIRECTIONS
None.

SEE ALSO
bsearch(), lsearch(), malloc(), strcmp(), tsearch(), the Base Definitions volume of
IEEE Std 1003.1-2001, <search.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
A note indicating that this function need not be reentrant is added to the DESCRIPTION.
**NAME**
htonl, htons, ntohl, ntohs — convert values between host and network byte order

**SYNOPSIS**
```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

**DESCRIPTION**
These functions shall convert 16-bit and 32-bit quantities between network byte order and host byte order.

On some implementations, these functions are defined as macros.

The `uint32_t` and `uint16_t` types are defined in `<inttypes.h>`.

**RETURN VALUE**
The `htonl()` and `htons()` functions shall return the argument value converted from host to network byte order.

The `ntohl()` and ` ntohs()` functions shall return the argument value converted from network to host byte order.

**ERRORS**
No errors are defined.

**EXAMPLES**
None.

**APPLICATION USAGE**
These functions are most often used in conjunction with IPv4 addresses and ports as returned by `gethostent()` and ` getservent()`. 

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`endhostent()`, `endservent()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<inttypes.h>`, `<arpa/inet.h>`

**CHANGE HISTORY**
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
hypot, hypotf, hypotl — Euclidean distance function

SYNOPSIS
#include <math.h>

double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);

DESCRIPTION
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

These functions shall compute the value of the square root of \( x^2 + y^2 \) without undue overflow or
underflow.

An application wishing to check for error situations should set \texttt{errno} to zero and call
\texttt{fcentercept(FE_ALL_EXCEPT)} before calling these functions. On return, if \texttt{errno} is non-zero or
\texttt{fetestexcept(\texttt{FE_INVALID} | \texttt{FE_DIVBYZERO} | \texttt{FE_OVERFLOW} | \texttt{FE_UNDERFLOW})} is non-
zero, an error has occurred.

RETURN VALUE
Upon successful completion, these functions shall return the length of the hypotenuse of a
right-angled triangle with sides of length \( x \) and \( y \).

If the correct value would cause overflow, a range error shall occur and \texttt{hypot()}, \texttt{hypotf()}, and
\texttt{hypotl()} shall return the value of the macro \texttt{HUGE_VAL}, \texttt{HUGE_VALF}, and \texttt{HUGE_VALL},
respectively.

If \( x \) or \( y \) is \( \pm \infty \), \( +\infty \) shall be returned (even if one of \( x \) or \( y \) is NaN).

If \( x \) or \( y \) is NaN, and the other is not \( \pm \infty \), a NaN shall be returned.

If both arguments are subnormal and the correct result is subnormal, a range error may occur
and the correct result is returned.

ERRORS
These functions shall fail if:

Range Error The result overflows.

If the integer expression (\texttt{math_errhandling} \& \texttt{MATH_ERRNO}) is non-zero,
then \texttt{errno} shall be set to \texttt{[ERANGE]}. If the integer expression
\texttt{math_errhandling} \& \texttt{MATH_ERREXCEPT} is non-zero, then the overflow
floating-point exception shall be raised.

These functions may fail if:

Range Error The result underflows.

If the integer expression (\texttt{math_errhandling} \& \texttt{MATH_ERRNO}) is non-zero,
then \texttt{errno} shall be set to \texttt{[ERANGE]}. If the integer expression
\texttt{math_errhandling} \& \texttt{MATH_ERREXCEPT} is non-zero, then the underflow
floating-point exception shall be raised.
**EXAMPLES**

None.

**APPLICATION USAGE**

\( \text{hypot}(x, y), \text{hypot}(y, x), \) and \( \text{hypot}(x, -y) \) are equivalent.

\( \text{hypot}(x, \pm 0) \) is equivalent to \( \text{fabs}(x) \).

Underflow only happens when both \( x \) and \( y \) are subnormal and the (inexact) result is also subnormal.

These functions take precautions against overflow during intermediate steps of the computation.

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.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

\text{feclearexcept}(), \text{fetestexcept}(), \text{isnan}(), \text{sqrt}(), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

**CHANGE HISTORY**

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

**Issue 5**

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

**Issue 6**

The \text{hypot}() function is no longer marked as an extension.

The \text{hypotf}() and \text{hypotl}() functions are added for alignment with the ISO/IEC 9899:1999 standard.

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

NAME

iconv — codeset conversion function

SYNOPSIS

XSI
#include <iconv.h>

size_t iconv(iconv_t cd, char **restrict inbuf,  
size_t *restrict inbytesleft, char **restrict outbuf,  
size_t *restrict outbytesleft);

DESCRIPTION

The iconv() function shall convert the sequence of characters from one codeset, in the array
specified by inbuf, into a sequence of corresponding characters in another codeset, in the array
specified by outbuf. The codesets are those specified in the iconv_open() call that returned the
conversion descriptor, cd. The inbuf argument points to a variable that points to the first
character in the input buffer and inbytesleft indicates the number of bytes to the end of the buffer
to be converted. The outbuf argument points to a variable that points to the first available byte in
the output buffer and outbytesleft indicates the number of the available bytes to the end of the
buffer.

For state-dependent encodings, the conversion descriptor cd is placed into its initial shift state by
a call for which inbuf is a null pointer, or for which inbuf points to a null pointer. When iconv() is
called in this way, and if outbuf is not a null pointer or a pointer to a null pointer, and outbytesleft
points to a positive value, iconv() shall place, into the output buffer, the byte sequence to change
the output buffer to its initial shift state. If the output buffer is not large enough to hold the
entire reset sequence, iconv() shall fail and set errno to [E2BIG]. Subsequent calls with inbuf as
other than a null pointer or a pointer to a null pointer cause the conversion to take place from
the current state of the conversion descriptor.

If a sequence of input bytes does not form a valid character in the specified codeset, conversion
shall stop after the previous successfully converted character. If the input buffer ends with an
incomplete character or shift sequence, conversion shall stop after the previous successfully
converted bytes. If the output buffer is not large enough to hold the entire converted input,
conversion shall stop just prior to the input bytes that would cause the output buffer to
overflow. The variable pointed to by inbuf shall be updated to point to the byte following the last
byte successfully used in the conversion. The value pointed to by inbytesleft shall be
decrementated to reflect the number of bytes still not converted in the input buffer. The variable
pointed to by outbuf shall be updated to point to the byte following the last byte of converted
output data. The value pointed to by outbytesleft shall be decremented to reflect the number of
bytes still available in the output buffer. For state-dependent encodings, the conversion
descriptor shall be updated to reflect the shift state in effect at the end of the last successfully
converted byte sequence.

If iconv() encounters a character in the input buffer that is valid, but for which an identical
character does not exist in the target codeset, iconv() shall perform an implementation-defined
conversion on this character.

RETURN VALUE

The iconv() function shall update the variables pointed to by the arguments to reflect the extent
of the conversion and return the number of non-identical conversions performed. If the entire
string in the input buffer is converted, the value pointed to by inbytesleft shall be 0. If the input
conversion is stopped due to any conditions mentioned above, the value pointed to by inbytesleft
shall be non-zero and errno shall be set to indicate the condition. If an error occurs, iconv() shall
return (size_t)−1 and set errno to indicate the error.
The `iconv()` function shall fail if:

- **[EILSEQ]** Input conversion stopped due to an input byte that does not belong to the input codeset.
- **[E2BIG]** Input conversion stopped due to lack of space in the output buffer.
- **[EINVAL]** Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.

The `iconv()` function may fail if:

- **[EBADF]** The `cd` argument is not a valid open conversion descriptor.

**EXAMPLES**

None.

**APPLICATION USAGE**

The `inbuf` argument indirectly points to the memory area which contains the conversion input data. The `outbuf` argument indirectly points to the memory area which is to contain the result of the conversion. The objects indirectly pointed to by `inbuf` and `outbuf` are not restricted to containing data that is directly representable in the ISO C standard language `char` data type. The type of `inbuf` and `outbuf`, `char **`, does not imply that the objects pointed to are interpreted as null-terminated C strings or arrays of characters. Any interpretation of a byte sequence that represents a character in a given character set encoding scheme is done internally within the codeset converters. For example, the area pointed to indirectly by `inbuf` and/or `outbuf` can contain all zero octets that are not interpreted as string terminators but as coded character data according to the respective codeset encoding scheme. The type of the data (`char`, `short`, `long`, and so on) read or stored in the objects is not specified, but may be inferred for both the input and output data by the converters determined by the `fromcode` and `tocode` arguments of `iconv_open()`.

Regardless of the data type inferred by the converter, the size of the remaining space in both input and output objects (the `inbytesleft` and `outbytesleft` arguments) is always measured in bytes.

For implementations that support the conversion of state-dependent encodings, the conversion descriptor must be able to accurately reflect the shift-state in effect at the end of the last successful conversion. It is not required that the conversion descriptor itself be updated, which would require it to be a pointer type. Thus, implementations are free to implement the descriptor as a handle (other than a pointer type) by which the conversion information can be accessed and updated.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`iconv_open()`, `iconv_close()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<iconv.h>`

**CHANGE HISTORY**


**Issue 6**

The SYNOPSIS has been corrected to align with the `<iconv.h>` reference page.

The `restrict` keyword is added to the `iconv()` prototype for alignment with the ISO/IEC 9899:1999 standard.
iconv_close()

NAME
iconv_close — codeset conversion deallocation function

SYNOPSIS
XSI
#include <iconv.h>

int iconv_close(iconv_t cd);

DESCRIPTION
The iconv_close() function shall deallocate the conversion descriptor cd and all other associated
resources allocated by iconv_open().

If a file descriptor is used to implement the type iconv_t, that file descriptor shall be closed.

RETURN VALUE
Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned and errno set to
indicate the error.

ERRORS
The iconv_close() function may fail if:
[EBADF] The conversion descriptor is invalid.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
iconv(), iconv_open(), the Base Definitions volume of IEEE Std 1003.1-2001, <iconv.h>

CHANGE HISTORY
NAME
iconv_open — codeset conversion allocation function

SYNOPSIS
XSI
#include <iconv.h>

iconv_t iconv_open(const char * tocode, const char * fromcode);

DESCRIPTION
The iconv_open() function shall return a conversion descriptor that describes a conversion from
the codeset specified by the string pointed to by the fromcode argument to the codeset specified
by the string pointed to by the tocode argument. For state-dependent encodings, the conversion
descriptor shall be in a codeset-dependent initial shift state, ready for immediate use with
iconv().

Settings of fromcode and tocode and their permitted combinations are implementation-defined.

A conversion descriptor shall remain valid until it is closed by iconv_close() or an implicit close.

If a file descriptor is used to implement conversion descriptors, the FD_CLOEXEC flag shall be
set; see <fcntl.h>.

RETURN VALUE
Upon successful completion, iconv_open() shall return a conversion descriptor for use on
subsequent calls to iconv(). Otherwise, iconv_open() shall return (iconv_t)-1 and set errno to
indicate the error.

ERRORS
The iconv_open() function may fail if:

[EMFILE] [OPEN_MAX] file descriptors are currently open in the calling process.
[ENFILE] Too many files are currently open in the system.
[ENOMEM] Insufficient storage space is available.
[EINVAL] The conversion specified by fromcode and tocode is not supported by the
implementation.

EXAMPLES
None.

APPLICATION USAGE
Some implementations of iconv_open() use malloc() to allocate space for internal buffer areas.
The iconv_open() function may fail if there is insufficient storage space to accommodate these
buffers.

Conforming applications must assume that conversion descriptors are not valid after a call to
one of the exec functions.

Application developers should consult the system documentation to determine the supported
codesets and their naming schemes.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO
iconv(), iconv_close(), the Base Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>, <iconv.h>

CHANGE HISTORY
NAME
if_freenameindex — free memory allocated by if_nameindex

SYNOPSIS
#include <net/if.h>
void if_freenameindex(struct if_nameindex *ptr);

DESCRIPTION
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.

RETURN VALUE
None.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getsockopt(), if_indextoname(), if_nameindex(), if_nametoindex(), setsockopt(), the Base Definitions volume of IEEE Std 1003.1-2001, <net/if.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
if_indextoname()

NAME
if_indextoname — map a network interface index to its corresponding name

SYNOPSIS
#include <net/if.h>
char *if_indextoname(unsigned ifindex, char *ifname);

DESCRIPTION
The if_indextoname() function shall map an interface index to its corresponding name.

When this function is called, ifname shall point to a buffer of at least {IFNAMSIZ} bytes. The
function shall place in this buffer the name of the interface with index ifindex.

RETURN VALUE
If ifindex is an interface index, then the function shall return the value supplied in ifname, which
points to a buffer now containing the interface name. Otherwise, the function shall return a
NULL pointer and set errno to indicate the error.

ERRORS
The if_indextoname() function shall fail if:

[ENXIO] The interface does not exist.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getsockopt(), if_freenameindex(), if_nameindex(), if_nametoindex(), setsockopt(), the Base
Definitions volume of IEEE Std 1003.1-2001, <net/if.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
if_nameindex — return all network interface names and indexes

SYNOPSIS
#include <net/if.h>
struct if_nameindex *if_nameindex(void);

DESCRIPTION
The if_nameindex() function shall return an array of if_nameindex structures, one structure per
interface. The end of the array is indicated by a structure with an if_index field of zero and an
if_name field of NULL.
Applications should call if_frenameindex() to release the memory that may be dynamically
allocated by this function, after they have finished using it.

RETURN VALUE
An array of structures identifying local interfaces. A NULL pointer is returned upon an error,
with errno set to indicate the error.

ERRORS
The if_nameindex() function may fail if:
[ENOBUFS] Insufficient resources are available to complete the function.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getsockopt(), if_frenameindex(), if_indextoname(), if_nametoindex(), setsockopt(), the Base
Definitions volume of IEEE Std 1003.1-2001, <net/if.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
if_nametoindex — map a network interface name to its corresponding index

SYNOPSIS
#include <net/if.h>
unsigned if_nametoindex(const char *ifname);

DESCRIPTION
The if_nametoindex() function shall return the interface index corresponding to name ifname.

RETURN VALUE
The corresponding index if ifname is the name of an interface; otherwise, zero.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getsockopt(), if_freenameindex(), if_indextoname(), if_nameindex(), setsockopt(), the Base Definitions volume of IEEE Std 1003.1-2001, <net/if.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
ilogb, ilogbf, ilogbl — return an unbiased exponent

SYNOPSIS
#include <math.h>

int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);

DESCRIPTION
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.

These functions shall return the exponent part of their argument \( x \). Formally, the return value is the integral part of \( \log_r |x| \) as a signed integral value, for non-zero \( x \), where \( r \) is the radix of the machine's floating-point arithmetic, which is the value of FLT_RADIX defined in <float.h>.

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

RETURN VALUE
Upon successful completion, these functions shall return the exponent part of \( x \) as a signed integer value. They are equivalent to calling the corresponding \texttt{logb()} function and casting the returned value to type \texttt{int}.

If \( x \) is 0, a domain error shall occur, and the value FP_ILOGB0 shall be returned.

If \( x \) is \( \pm \)Inf, a domain error shall occur, and the value \{INT_MAX\} shall be returned.

If \( x \) is a NaN, a domain error shall occur, and the value FP_ILOGBNAN shall be returned.

If the correct value is greater than \{INT_MAX\}, \{INT_MAX\} shall be returned and a domain error shall occur.

If the correct value is less than \{INT_MIN\}, \{INT_MIN\} shall be returned and a domain error shall occur.

ERRORS
These functions shall fail if:

- Domain Error The \( x \) argument is zero, NaN, or \( \pm \)Inf, or the correct value is not representable as an integer.

If the integer expression (\texttt{math_errhandling} \& \texttt{MATH_ERRNO}) is non-zero, then \texttt{errno} shall be set to [EDOM]. If the integer expression (\texttt{math_errhandling} \& \texttt{MATH_ERREXCEPT}) is non-zero, then the invalid floating-point exception shall be raised.
**EXAMPLES**
None.

**APPLICATION USAGE**
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.

**RATIONALE**
The errors come from taking the expected floating-point value and converting it to `int`, which is an invalid operation in IEEE Std 754-1985 (since overflow, infinity, and NaN are not representable in a type `int`), so should be a domain error.

There are no known implementations that overflow. For overflow to happen, `{INT_MAX}` must be less than \( \text{LDBL\_MAX\_EXP} \* \log_2(\text{FLT\_RADIX}) \) or `{INT_MIN}` must be greater than \( \text{LDBL\_MIN\_EXP} \* \log_2(\text{FLT\_RADIX}) \) if subnormals are not supported, or `{INT_MIN}` must be greater than \( \text{(LDBL\_MIN\_EXP-\text{LDBL\_MANT\_DIG})} \* \log_2(\text{FLT\_RADIX}) \) if subnormals are supported.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`feclearexcept()`, `fetestexcept()`, `logb()`, `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>`

**CHANGE HISTORY**
First released in Issue 4, Version 2.

**Issue 5**
Moved from X/OPEN UNIX extension to BASE.

**Issue 6**
The `ilogb()` function is no longer marked as an extension.
The `ilogbf()` and `ilogbl()` functions are added for alignment with the ISO/IEC 9899:1999 standard.
The RETURN VALUE section is revised for alignment with the ISO/IEC 9899:1999 standard.
XSI extensions are marked.
NAME
imaxabs — return absolute value

SYNOPSIS
#include <inttypes.h>

intmax_t imaxabs(intmax_t j);

DESCRIPTION
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 imaxabs() function shall compute the absolute value of an integer j. If the result cannot be represented, the behavior is undefined.

RETURN VALUE
The imaxabs() function shall return the absolute value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
The absolute value of the most negative number cannot be represented in two's complement.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
imaxdiv(), the Base Definitions volume of IEEE Std 1003.1-2001, <inttypes.h>

CHANGE HISTORY
NAME
imaxdiv — return quotient and remainder

SYNOPSIS
#include <inttypes.h>
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);

DESCRIPTION
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 imaxdiv() function shall compute numer / denom and numer % denom in a single operation.

RETURN VALUE
The imaxdiv() function shall return a structure of type imaxdiv_t, comprising both the quotient and the remainder. The structure shall contain (in either order) the members quot (the quotient) and rem (the remainder), each of which has type intmax_t.
If either part of the result cannot be represented, the behavior is undefined.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
imaxabs(), the Base Definitions volume of IEEE Std 1003.1-2001, <inttypes.h>

CHANGE HISTORY
index()

NAME
index — character string operations (LEGACY)

SYNOPSIS
XSI
#include <strings.h>

char *index(const char *s, int c);

DESCRIPTION
The index() function shall be equivalent to strchr().

RETURN VALUE
See strchr().

ERRORS
See strchr().

EXAMPLES
None.

APPLICATION USAGE
The strchr() function is preferred over this function.
For maximum portability, it is recommended to replace the function call to index() as follows:
#define index(a,b) strchr((a),(b))

RATIONALE
None.

FUTURE DIRECTIONS
This function may be withdrawn in a future version.

SEE ALSO
strchr(), the Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
This function is marked LEGACY.
NAME
inet_addr, inet_ntoa — IPv4 address manipulation

SYNOPSIS
#include <arpa/inet.h>

in_addr_t inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);

DESCRIPTION
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.

The inet_ntoa() function shall convert the Internet host address specified by in to a string in the
Internet standard dot notation.

The inet_ntoa() function need not be reentrant. A function that is not required to be reentrant is
not required to be thread-safe.

All Internet addresses shall be returned in network order (bytes ordered from left to right).

Values specified using IPv4 dotted decimal notation take one of the following forms:

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.

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".

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".

a           When only one part is given, the value shall be stored directly in the network
address without any byte rearrangement.

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).

RETURN VALUE
Upon successful completion, inet_addr() shall return the Internet address. Otherwise, it shall
return (in_addr_t)(-1).

The inet_ntoa() function shall return a pointer to the network address in Internet standard dot
notation.

ERRORS
No errors are defined.
**EXAMPLES**

None.

**APPLICATION USAGE**

The return value of `inet_ntoa()` may point to static data that may be overwritten by subsequent calls to `inet_ntoa()`.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`endhostent()`, `endnetent()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<arpa/inet.h>`

**CHANGE HISTORY**

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
inet_ntop, inet_pton — convert IPv4 and IPv6 addresses between binary and text form

SYNOPSIS
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *restrict src,
           char *restrict dst, socklen_t size);

int inet_pton(int af, const char *restrict src, void *restrict dst);

DESCRIPTION
The inet_ntop() function shall convert a numeric address into a text string suitable for presentation. The af argument shall specify the family of the address. This can be AF_INET or AF_INET6. The src argument points to a buffer holding an IPv4 address if the af argument is AF_INET, or an IPv6 address if the af argument is AF_INET6. The dst argument points to a buffer where the function stores the resulting text string; it shall not be NULL. The size argument specifies the size of this buffer, which shall be large enough to hold the text string (INET_ADDRSTRLEN characters for IPv4,INET6_ADDRSTRLEN characters for IPv6).

The inet_pton() function shall convert an address in its standard text presentation form into its numeric binary form. The af argument shall specify the family of the address. The AF_INET and AF_INET6 address families shall be supported. The src argument points to the string being passed in. The dst argument points to a buffer into which the function stores the numeric address; this shall be large enough to hold the numeric address (32 bits for AF_INET, 128 bits for AF_INET6).

If the af argument of inet_pton() is AF_INET, the src string shall be in the standard IPv4 dotted-decimal form:

ddd.ddd.ddd.ddd

where "ddd" is a one to three digit decimal number between 0 and 255 (see inet_addr()). The inet_pton() function does not accept other formats (such as the octal numbers, hexadecimal numbers, and fewer than four numbers that inet_addr() accepts).

If the af argument of inet_pton() is AF_INET6, the src string shall be in one of the following standard IPv6 text forms:

1. The preferred form is "x:x:x:x:x:x:x", where the ’x’s are the hexadecimal values of the eight 16-bit pieces of the address. Leading zeros in individual fields can be omitted, but there shall be at least one numeral in every field.

2. A string of contiguous zero fields in the preferred form can be shown as "::". The "::" can only appear once in an address. Unspecified addresses ("0:0:0:0:0:0:0:0") may be represented simply as "::".

3. A third form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes is "x:x:x:x:x:x:d.d.d", where the ’x’s are the hexadecimal values of the six high-order 16-bit pieces of the address, and the ’d’s are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation).

Note: A more extensive description of the standard representations of IPv6 addresses can be found in RFC 2373.
inet_ntop()  

RETURN VALUE

The inet_ntop() function shall return a pointer to the buffer containing the text string if the conversion succeeds, and NULL otherwise, and set errno to indicate the error.

The inet_pton() function shall return 1 if the conversion succeeds, with the address pointed to by dst in network byte order. It shall return 0 if the input is not a valid IPv4 dotted-decimal string or a valid IPv6 address string, or −1 with errno set to [EAFNOSUPPORT] if the af argument is unknown.

ERRORS

The inet_ntop() and inet_pton() functions shall fail if:

[EAFNOSUPPORT] The af argument is invalid.

[ENOSPC] The size of the inet_ntop() result buffer is inadequate.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-2001, <arpa/inet.h>

CHANGE HISTORY

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

IPv6 extensions are marked.

The restrict keyword is added to the inet_ntop() and inet_pton() prototypes for alignment with the ISO/IEC 9899:1999 standard.
NAME
initstate, random, setstate, srandom — pseudo-random number functions

SYNOPSIS
#include <stdlib.h>

char *initstate(unsigned seed, char *state, size_t size);
long random(void);
char *setstate(const char *state);
void srandom(unsigned seed);

DESCRIPTION
The random() function shall use a non-linear additive feedback random-number generator employing a default state array size of 31 long integers to return successive pseudo-random numbers in the range from 0 to 2^{31}−1. The period of this random-number generator is approximately 16 x (2^{31}−1). The size of the state array determines the period of the random-number generator. Increasing the state array size shall increase the period.

With 256 bytes of state information, the period of the random-number generator shall be greater than 2^{69}.

Like rand(), random() shall produce by default a sequence of numbers that can be duplicated by calling srandom() with 1 as the seed.

The srandom() function shall initialize the current state array using the value of seed.

The initstate() and setstate() functions handle restarting and changing random-number generators. The initstate() function allows a state array, pointed to by the state argument, to be initialized for future use. The size argument, which specifies the size in bytes of the state array, shall be used by initstate() to decide what type of random-number generator to use; the larger the state array, the more random the numbers. Values for the amount of state information are 8, 32, 64, 128, and 256 bytes. Other values greater than 8 bytes are rounded down to the nearest one of these values. If initstate() is called with 8≤size<32, then random() shall use a simple linear congruential random number generator. The seed argument specifies a starting point for the random-number sequence and provides for restarting at the same point. The initstate() function shall return a pointer to the previous state information array.

If initstate() has not been called, then random() shall behave as though initstate() had been called with seed=1 and size=128.

Once a state has been initialized, setstate() allows switching between state arrays. The array defined by the state argument shall be used for further random-number generation until initstate() is called or setstate() is called again. The setstate() function shall return a pointer to the previous state array.

RETURN VALUE
If initstate() is called with size less than 8, it shall return NULL.

The random() function shall return the generated pseudo-random number.

The srandom() function shall not return a value.

Upon successful completion, initstate() and setstate() shall return a pointer to the previous state array; otherwise, a null pointer shall be returned.
initstate()

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
After initialization, a state array can be restarted at a different point in one of two ways:

1. The initstate() function can be used, with the desired seed, state array, and size of the array.

2. The setstate() function, with the desired state, can be used, followed by srandom() 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.

Although some implementations of random() have written messages to standard error, such implementations do not conform to IEEE Std 1003.1-2001.

Issue 5 restored the historical behavior of this function.

Threaded applications should use rand_r(), erand48(), nrand48(), or jrand48() instead of random() when an independent random number sequence in multiple threads is required.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
drand48(), rand(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

In the DESCRIPTION, the phrase “values smaller than 8” is replaced with “values greater than or equal to 8, or less than 32”, “size<8” is replaced with “8≤size<32”, and a new first paragraph is added to the RETURN VALUE section. A note is added to the APPLICATION USAGE indicating that these changes restore the historical behavior of the function.

Issue 6
In the DESCRIPTION, duplicate text “For values greater than or equal to 8 …” is removed.
NAME
insque, remque — insert or remove an element in a queue

SYNOPSIS
XSI
#include <search.h>

void insque(void *element, void *pred);
void remque(void *element);

DESCRIPTION
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.

The insque() function shall insert the element pointed to by element into a queue immediately after the element pointed to by pred.

The remque() function shall remove the element pointed to by element from a queue.

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.

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.

RETURN VALUE
The insque() and remque() functions do not return a value.

ERRORS
No errors are defined.

EXAMPLES
Creating a Linear Linked List
The following example creates a linear linked list.
#include <search.h>
...
struct myque element1;
struct myque element2;
char *data1 = "DATA1";
char *data2 = "DATA2";
...
element1.data = data1;
element2.data = data2;
insque (&element1, NULL);
insque (&element2, &element1);
Creating a Circular Linked List
The following example creates a circular linked list.

```c
#include <search.h>
...
struct myque element1;
struct myque element2;
char *data1 = "DATA1";
char *data2 = "DATA2";
...
element1.data = data1;
element2.data = data2;
element1.fwd = &element1;
element1.bck = &element1;
insque (&element2, &element1);
```

Removing an Element
The following example removes the element pointed to by `element1`.

```c
#include <search.h>
...
struct myque element1;
...
remque (&element1);
```

APPLICATION USAGE
The historical implementations of these functions described the arguments as being of type
`struct qelem *` rather than as being of type `void *` as defined here. In those implementations,
`struct qelem` was commonly defined in `<search.h>` as:

```c
struct qelem {
    struct qelem *q_forw;
    struct qelem *q_back;
};
```

Applications using these functions, however, were never able to use this structure directly since
it provided no room for the actual data contained in the elements. Most applications defined
structures that contained the two pointers as the initial elements and also provided space for, or
pointers to, the object’s data. Applications that used these functions to update more than one
type of table also had the problem of specifying two or more different structures with the same
name, if they literally used `struct qelem` as specified.

As described here, the implementations were actually expecting a structure type where the first
two members were forward and backward pointers to structures. With C compilers that didn’t
provide function prototypes, applications used structures as specified in the DESCRIPTION
above and the compiler did what the application expected.

If this method had been carried forward with an ISO C standard compiler and the historical
function prototype, most applications would have to be modified to cast pointers to the
structures actually used to be pointers to `struct qelem` to avoid compilation warnings. By
specifying `void *` as the argument type, applications do not need to change (unless they
specifically referenced `struct qelem` and depended on it being defined in `<search.h>`).
RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <search.h>

CHANGE HISTORY
First released in Issue 4, Version 2.
Moved from X/OPEN UNIX extension to BASE.
The DESCRIPTION is updated to avoid use of the term "must" for application requirements.
NAME
ioctl — control a STREAMS device (STREAMS)

SYNOPSIS
XSR
#include <stropts.h>
int ioctl(int fildes, int request, ... /* arg */);

DESCRIPTION
The ioctl() function shall perform a variety of control functions on STREAMS devices. For non-
STREAMS devices, the functions performed by this call are unspecified. The request argument
and an optional third argument (with varying type) shall be passed to and interpreted by the
appropriate part of the STREAM associated with fildes.

The fildes argument is an open file descriptor that refers to a device.

The request argument selects the control function to be performed and shall depend on the
STREAMS device being addressed.

The arg argument represents additional information that is needed by this specific STREAMS
device to perform the requested function. The type of arg depends upon the particular control
request, but it shall be either an integer or a pointer to a device-specific data structure.

The ioctl() commands applicable to STREAMS, their arguments, and error conditions that apply
to each individual command are described below.

The following ioctl() commands, with error values indicated, are applicable to all STREAMS
files:

I_PUSH Pushes the module whose name is pointed to by arg onto the top of the
current STREAM, just below the STREAM head. It then calls the open() function of the newly-pushed module.

The ioctl() function with the I_PUSH command shall fail if:

- [EINVAL] Invalid module name.
- [ENXIO] Hangup received on fildes.

I_POP Removes the module just below the STREAM head of the STREAM pointed to
by fildes. The arg argument should be 0 in an I_POP request.

The ioctl() function with the I_POP command shall fail if:

- [EINVAL] No module present in the STREAM.
- [ENXIO] Hangup received on fildes.

I_LOOK Retrieves the name of the module just below the STREAM head of the STREAM pointed to
by fildes, and places it in a character string pointed to by arg. The buffer pointed to by arg should be at least FMNAMESZ+1 bytes long,
where FMNAMESZ is defined in <stropts.h>.

The ioctl() function with the I_LOOK command shall fail if:

- [EINVAL] No module present in the STREAM.
- [ENXIO] Hangup received on fildes.

I_FLUSHFlushes read and/or write queues, depending on the value of arg. Valid arg
values are:
System Interfaces

ioctl()

2028  FLUSHR  Flush all read queues.
2029  FLUSHW  Flush all write queues.
2030  FLUSHRW Flush all read and all write queues.
2031  The ioctl() function with the I_FLUSH command shall fail if:
2032  [EINVAL] Invalid arg value.
2033  [EAGAIN] or [ENOSR]
2034  Unable to allocate buffers for flush message.
2035  [ENXIO]  Hangup received on fildes.
2036  I_FLUSHBAND Flushing a particular band of messages. The arg argument points to a bandinfo
2037  structure. The bi_flag member may be one of FLUSHR, FLUSHW, or
2038  FLUSHRW as described above. The bi_pri member determines the priority
2039  band to be flushed.
2040  I_SETSIG Requests that the STREAMS implementation send the SIGPOLL signal to the
2041  calling process when a particular event has occurred on the STREAM
2042  associated with fildes. I_SETSIG supports an asynchronous processing
2043  capability in STREAMS. The value of arg is a bitmask that specifies the events
2044  for which the process should be signaled. It is the bitwise-inclusive OR of any
2045  combination of the following constants:
2046  S_RDNORM  A normal (priority band set to 0) message has arrived at the
2047  head of a STREAM head read queue. A signal shall be
2048  generated even if the message is of zero length.
2049  S_RDBAND  A message with a non-zero priority band has arrived at the
2050  head of a STREAM head read queue. A signal shall be
2051  generated even if the message is of zero length.
2052  S_INPUT   A message, other than a high-priority message, has arrived
2053  at the head of a STREAM head read queue. A signal shall be
2054  generated even if the message is of zero length.
2055  S_HIPRI   A high-priority message is present on a STREAM head read
2056  queue. A signal shall be generated even if the message is of
2057  zero length.
2058  S_OUTPUT  The write queue for normal data (priority band 0) just
2059  below the STREAM head is no longer full. This notifies the
2060  process that there is room on the queue for sending (or
2061  writing) normal data downstream.
2062  S_WRNORM  Equivalent to S_OUTPUT.
2063  S_WRBAND  The write queue for a non-zero priority band just below the
2064  STREAM head is no longer full. This notifies the process
2065  that there is room on the queue for sending (or writing)
2066  priority data downstream.
2067  S_MSG     A STREAMS signal message that contains the SIGPOLL
2068  signal has reached the front of the STREAM head read
2069  queue.
2070  S_ERROR  Notification of an error condition has reached the STREAM
2071  head.
ioctl()

S_HANGUP  Notification of a hangup has reached the STREAM head.

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.

If arg is 0, the calling process shall be unregistered and shall not receive
further SIGPOLL signals for the stream associated with fildes.

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.

The ioctl() function with the I_SETSIG command shall fail if:

EINVAL  The value of arg is invalid.

EINVAL  The value of arg is 0 and the calling process is not registered
to receive the SIGPOLL signal.

EFAULT  There were insufficient resources to store the signal request.

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 arg, where the events are those specified in the description of I_SETSIG
above.

The ioctl() function with the I_GETSIG command shall fail if:

EINVAL  Process is not registered to receive the SIGPOLL signal.

I_FIND  Compares the names of all modules currently present in the STREAM to the
name pointed to by arg, and returns 1 if the named module is present in the
STREAM, or returns 0 if the named module is not present.

The ioctl() function with the I_FIND command shall fail if:

EINVAL  arg does not contain a valid module name.

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 getmsg().
except that this command does not remove the message from the queue. The
arg argument points to a strpeek structure.

The application shall ensure that the buflen member in the ctlbuf and databuf
structures is set to the number of bytes of control information and/or
data information, respectively, to retrieve. The flags member may be marked
RS_HIPRI or 0, as described by getmsg(). If the process sets flags to RS_HIPRI,
for example, I_PEEK shall only look for a high-priority message on the
STREAM head read queue.

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
flags 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 flags contains the value RS_HIPRI or 0.

I_SRDOPT  Sets the read mode using the value of the argument arg. Read modes are
described in read(). Valid arg flags are:
The bitwise-inclusive OR of RMSGD and RMSGN shall return [EINVAL]. The bitwise-inclusive OR of RNORM and either RMSGD or RMSGN shall result in the other flag overriding RNORM which is the default.

In addition, treatment of control messages by the STREAM head may be changed by setting any of the following flags in arg:

- **RPROTNORM** Fail `read()` with [EBADMSG] if a message containing a control part is at the front of the STREAM head read queue.
- **RPROTDAT** Deliver the control part of a message as data when a process issues a `read()`.
- **RPROTDIS** Discard the control part of a message, delivering any data portion, when a process issues a `read()`.

The `ioctl()` function with the I_SRDOPT command shall fail if:

- `[EINVAL]` The arg argument is not valid.

**I_GRDOPT**

Returns the current read mode setting, as described above, in an int pointed to by the argument arg. Read modes are described in `read()`.

**I_NREAD**

Counts the number of data bytes in the data part of the first message on the STREAM head read queue and places this value in the int pointed to by arg. The return value for the command shall be the number of messages on the STREAM head read queue. For example, if 0 is returned in arg, but the `ioctl()` return value is greater than 0, this indicates that a zero-length message is next on the queue.

**I_FDINSERT**

Creates a message from specified buffer(s), adds information about another STREAM, and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The arg argument points to a `strfdinsert` structure.

The application shall ensure that the len member in the ctbuf strbuf structure is set to the size of a t_uscalar_t plus the number of bytes of control information to be sent with the message. The fildes member specifies the file descriptor of the other STREAM, and the offset member, which must be suitably aligned for use as a t_uscalar_t, specifies the offset from the start of the control buffer where I_FDINSERT shall store a t_uscalar_t whose interpretation is specific to the STREAM end. The application shall ensure that the len member in the databuf strbuf structure is set to the number of bytes of data information to be sent with the message, or to 0 if no data part is to be sent.

The flags member specifies the type of message to be created. A normal message is created if flags is set to 0, and a high-priority message is created if flags is set to RS_HIPRI. For non-priority messages, I_FDINSERT shall block if the STREAM write queue is full due to internal flow control conditions. For priority messages, I_FDINSERT does not block on this condition. For non-priority messages, I_FDINSERT does not block when the write queue is full.
and O_NONBLOCK is set. Instead, it fails and sets errno to [EAGAIN].

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.

The ioctl() function with the I_FDINSERT command shall fail if:

- [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.
- [EAGAIN] or [ENOSR]
  Buffers cannot be allocated for the message that is to be created.
- [EINVAL]
  One of the following:
  - The fildes member of the strfdinsert structure is not a valid, open STREAM file descriptor.
  - The size of a t_uscalar_t plus offset is greater than the len member for the buffer specified through ctlbuf.
  - The offset member does not specify a properly-aligned location in the data buffer.
  - An undefined value is stored in flags.
- [ENXIO] Hangup received on the STREAM identified by either the fildes argument or the fildes member of the strfdinsert structure.
- [ERANGE] The len 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 len member for the buffer specified through databuf is larger than the maximum configured size of the data part of a message; or the len member for the buffer specified through ctlbuf is larger than the maximum configured size of the control part of a message.

I_STR Constructs an internal STREAMS ioctl() message from the data pointed to by arg, and sends that message downstream. This mechanism is provided to send ioctl() requests to downstream modules and drivers. It allows information to be sent with ioctl(), 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 errno set to [ETIME].

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.

To send requests downstream, the application shall ensure that arg points to a strioctl structure.
The `ic_cmd` member is the internal `ioctl()` command intended for a downstream module or driver and `ic_timeout` is the number of seconds

\((-1=\text{infinite}, 0=\text{use implementation-defined timeout interval}, >0=\text{as specified})\)

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).

The STREAM head shall convert the information pointed to by the `strioctl` structure to an internal `ioctl()` command message and send it downstream.

The `ioctl()` function with the I_STR command shall fail if:

- [EAGAIN] or [ENOSR]

  Unable to allocate buffers for the `ioctl()` message.

- [EINVAL]

  The `ic_len` member is less than 0 or larger than the maximum configured size of the data part of a message, or `ic_timeout` is less than −1.

- [ENXIO]

  Hangup received on `fildes`.

- [ETIME]

  A downstream `ioctl()` timed out before acknowledgement was received.

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.

**I_SWROPT**

Sets the write mode using the value of the argument `arg`. Valid bit settings for `arg` are:

- **SNDZERO**

  Send a zero-length message downstream when a `write()` of 0 bytes occurs. To not send a zero-length message when a `write()` of 0 bytes occurs, the application shall ensure that this bit is not set in `arg` (for example, `arg` would be set to 0).

The `ioctl()` function with the I_SWROPT command shall fail if:

- [EINVAL]

  `arg` is not the above value.

**I_GWROPT**

Returns the current write mode setting, as described above, in the `int` that is pointed to by the argument `arg`.

**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 `fildes` containing this reference, together with the user ID and group ID of the calling process.

The `ioctl()` function with the I_SENDFD command shall fail if:

- [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.
The `ioctl()` function with the I_RECVFD command shall fail if:

- [EAGAIN] A message is not present at the STREAM head read queue and the O_NONBLOCK flag is set.
- [EBADMSG] The message at the STREAM head read queue is not a message containing a passed file descriptor.
- [EMFILE] The process has the maximum number of file descriptors currently open that it is allowed.
- [ENXIO] Hangup received on `fildes`.

The `ioctl()` function with the I_LIST command shall fail if:

- [EINVAL] The `sl_nmods` member is less than 1.
- [EAGAIN] or [ENOSR] Unable to allocate buffers.

The `ioctl()` function with the I_ATMARK command shall fail if the message at the head of the STREAM head read queue is marked by some module downstream. The `arg` argument determines
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:

ANYMARK Check if the message is marked.

LASTMARK Check if the message is the last one marked on the queue.

The bitwise-inclusive OR of the flags ANYMARK and LASTMARK is
permitted.

The return value shall be 1 if the mark condition is satisfied; otherwise, the
value shall be 0.

The `ioctl()` function with the I_ATMARK command shall fail if:

EINVAL Invalid arg value.

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. arg should be of type int.

The `ioctl()` function with the I_CKBAND command shall fail if:

EINVAL Invalid arg value.

I_GETBAND Returns the priority band of the first message on the STREAM head read
queue in the integer referenced by arg.

The `ioctl()` function with the I_GETBAND command shall fail if:

ENODATA No message on the STREAM head read queue.

I_CANPUT Checks if a certain band is writable. arg 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.

The `ioctl()` function with the I_CANPUT command shall fail if:

EINVAL Invalid arg value.

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 arg 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.

The `ioctl()` function with the I_SETCLTIME command shall fail if:

EINVAL Invalid arg value.

I_GETCLTIME Returns the close time delay in the integer pointed to by arg.
Multiplexed STREAMS Configurations

The following commands are used for connecting and disconnecting multiplexed STREAMS configurations. These commands use an implementation-defined default timeout interval.

**I_LINK**

Connects two STREAMs, where *fildes* is the file descriptor of the STREAM connected to the multiplexing driver, and *arg* is the file descriptor of the STREAM connected to another driver. The STREAM designated by *arg* 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.

The `ioctl()` function with the I_LINK command shall fail if:

- [ENXIO] Hangup received on *fildes*.
- [ETIME] Timeout before acknowledgement message was received at STREAM head.
- [EAGAIN] or [ENOSR] Unable to allocate STREAMS storage to perform the I_LINK.
- [EBADF] The *arg* argument is not a valid, open file descriptor.
- [EINVAL] The *fildes* argument does not support multiplexing; or *arg* 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.

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 *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_LINK fails with errno set to the value in the message.

**I_UNLINK**

Disconnects the two STREAMs specified by *fildes* and *arg*. *fildes* is the file descriptor of the STREAM connected to the multiplexing driver. The *arg* argument is the multiplexer ID number that was returned by the I_LINK `ioctl()` command when a STREAM was connected downstream from the multiplexing driver. If *arg* is MUXID_ALL, then all STREAMs that were connected to *fildes* shall be disconnected. As in I_LINK, this command requires acknowledgement.

The `ioctl()` function with the I_UNLINK command shall fail if:

- [ENXIO] Hangup received on *fildes*.
- [ETIME] Timeout before acknowledgement message was received at STREAM head.
- [EAGAIN] or [ENOSR] Unable to allocate buffers for the acknowledgement message.
- [EINVAL] Invalid multiplexer ID number.
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 fildes. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_UNLINK shall fail with errno set to the value in the message.

I_PLINK

Creates a persistent connection between two STREAMs, where fildes is the file descriptor of the STREAM connected to the multiplexing driver, and arg 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 fildes associated with the upper STREAM to the multiplexing driver is closed. The STREAM designated by arg 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.

The ioctl() function with the I_PLINK command shall fail if:

- [ENXIO] Hangup received on fildes.
- [ETIME] Timeout before acknowledgement message was received at STREAM head.
- [EAGAIN] or [ENOSR] Unable to allocate STREAMS storage to perform the I_PLINK.
- [EBADF] The arg argument is not a valid, open file descriptor.
- [EINVAL] The fildes argument does not support multiplexing; or arg 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.

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 fildes. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_PLINK shall fail with errno set to the value in the message.

I_PUNLINK

Disconnects the two STREAMs specified by fildes and arg from a persistent connection. The fildes argument is the file descriptor of the STREAM connected to the multiplexing driver. The arg argument is the multiplexer ID number that was returned by the I_PLINK ioctl() command when a STREAM was connected downstream from the multiplexing driver. If arg is MUXID_ALL, then all STREAMs which are persistent connections to fildes shall be disconnected. As in I_PLINK, this command requires the multiplexing driver to acknowledge the request.

The ioctl() function with the I_PUNLINK command shall fail if:

- [ENXIO] Hangup received on fildes.
- [ETIME] Timeout before acknowledgement message was received at STREAM head.
[EAGAIN] or [ENOSR]
Unable to allocate buffers for the acknowledgement message.

[EINVAL] Invalid multiplexer ID number.

An I_PUNLINK 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 fildes. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_PUNLINK shall fail with errno set to the value in the message.

RETURN VALUE
Upon successful completion, ioctl() shall return a value other than −1 that depends upon the STREAMS device control function. Otherwise, it shall return −1 and set errno to indicate the error.

ERRORS
Under the following general conditions, ioctl() shall fail if:

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

[EINTR] A signal was caught during the ioctl() operation.

[EINVAL] The STREAM or multiplexer referenced by fildes is linked (directly or indirectly) downstream from a multiplexer.

If an underlying device driver detects an error, then ioctl() shall fail if:

[EINVAL] The request or arg argument is not valid for this device.

[EIO] Some physical I/O error has occurred.

[ENOTTY] The fildes argument is not associated with a STREAMS device that accepts control functions.

[ENXIO] The request and arg arguments are valid for this device driver, but the service requested cannot be performed on this particular sub-device.

[ENODEV] The fildes argument refers to a valid STREAMS device, but the corresponding device driver does not support the ioctl() function.

If a STREAM is connected downstream from a multiplexer, any ioctl() command except I_UNLINK and I_PUNLINK shall set errno to [EINVAL].

EXAMPLES
None.

APPLICATION USAGE
The implementation-defined timeout interval for STREAMS has historically been 15 seconds.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.6 (on page 38), close(), fcntl(), getmsg(), open(), pipe(), poll(), putmsg(), read(), sigaction(), write(), the Base Definitions volume of IEEE Std 1003.1-2001, <stropts.h>
CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
The Open Group Corrigendum U028/4 is applied, correcting text in the I_FDINSERT [EINVAL] case to refer to ctlbuf.
This function is marked as part of the XSI STREAMS Option Group.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
isalnum — test for an alphanumeric character

SYNOPSIS
#include <ctype.h>
int isalnum(int c);

DESCRIPTION
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

The isalnum() function shall test whether c is a character of class alpha or digit in the program's
current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

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.

RETURN VALUE
The isalnum() function shall return non-zero if c is an alphanumeric character; otherwise, it shall
return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
To ensure applications portability, especially across natural languages, only this function and
those listed in the SEE ALSO section should be used for character classification.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(),
setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>,
<stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The DESCRIPTION is updated to avoid use of the term "must" for application requirements.
isalpha() — test for an alphabetic character

#include <ctype.h>

int isalpha(int c);

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.

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.

The isalpha() function shall return non-zero if c is an alphabetic character; otherwise, it shall return 0.

No errors are defined.

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

None.

isalnum(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>, <stdio.h>

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

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

NAME
isascii — test for a 7-bit US-ASCII character

SYNOPSIS
XSI

#include <ctype.h>

int isascii(int c);

DESCRIPTION
The isascii() function shall test whether c is a 7-bit US-ASCII character code.
The isascii() function is defined on all integer values.

RETURN VALUE
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.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <ctype.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
isastream — test a file descriptor (STREAMS)

SYNOPSIS
#include <stropts.h>

int isastream(int fildes);

DESCRIPTION
The isastream() function shall test whether fildes, an open file descriptor, is associated with a
STREAMS-based file.

RETURN VALUE
Upon successful completion, isastream() shall return 1 if fildes refers to a STREAMS-based file
and 0 if not. Otherwise, isastream() shall return −1 and set errno to indicate the error.

ERRORS
The isastream() function shall fail if:
[EBADF] The fildes argument is not a valid open file descriptor.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <stropts.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
isatty()

NAME
isatty — test for a terminal device

SYNOPSIS
#include <unistd.h>

int isatty(int fildes);

DESCRIPTION
The isatty() function shall test whether fildes, an open file descriptor, is associated with a terminal device.

RETURN VALUE
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.

ERRORS
The isatty() function may fail if:

[EBADF] The fildes argument is not a valid open file descriptor.
[ENOTTY] The fildes argument is not associated with a terminal.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

• The optional setting of errno to indicate an error is added.
• The [EBADF] and [ENOTTY] optional error conditions are added.
NAME
isblank — test for a blank character

SYNOPSIS
#include <ctype.h>
int isblank(int c);

DESCRIPTION
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 isblank() function shall test whether c is a character of class blank in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

The c argument is a type int, the value of which the application shall ensure is a character 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.

RETURN VALUE
The isblank() function shall return non-zero if c is a <blank>; otherwise, it shall return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

CHANGE HISTORY
NAME
iscntrl — test for a control character

SYNOPSIS
#include <ctype.h>
int iscntrl(int c);

DESCRIPTION
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 iscntrl() function shall test whether c is a character of class cntrl in the program’s current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

The c argument is a type int, the value of which the application shall ensure is a character 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.

RETURN VALUE
The iscntrl() function shall return non-zero if c is a control character; otherwise, it shall return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
isalnum(), isalpha(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
isdigit — test for a decimal digit

SYNOPSIS
#include <ctype.h>
int isdigit(int c);

DESCRIPTION
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.

The isdigit() function shall test whether c is 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.

The c argument is an int, the value of which the application shall ensure is a character 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.

RETURN VALUE
The isdigit() function shall return non-zero if c is a decimal digit; otherwise, it shall return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
isalnum(), isalpha(), iscntrl(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
isfinite()

NAME
isfinite — test for finite value

SYNOPSIS
#include <math.h>
int isfinite(real-floating x);

DESCRIPTION
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 isfinite() macro shall determine whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). 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.

RETURN VALUE
The isfinite() macro shall return a non-zero value if and only if its argument has a finite value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fpclassify(), isnanf(), isnanl(), isinf(), isnormal(), signbit(), the Base Definitions volume of IEEE Std 1003.1-2001 <math.h>

CHANGE HISTORY
NAME

isgraph — test for a visible character

SYNOPSIS

#include <ctype.h>

int isgraph(int c);

DESCRIPTION

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

The isgraph() function shall test whether c is 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.

The c argument is an int, the value of which the application shall ensure is a character
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.

RETURN VALUE

The isgraph() function shall return non-zero if c is a character with a visible representation;
otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and
those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum(), isalpha(), iscntrl(), isdigit(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(),
setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

CHANGE HISTORY

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

Issue 6

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
isgreater() — test if x greater than y

NAME

isgreater — test if x greater than y

SYNOPSIS

```
#include <math.h>

int isgreater(real-floating x, real-floating y);
```

DESCRIPTION

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 isgreater() macro shall determine whether its first argument is greater than its second argument. The value of isgreater(x, y) shall be equal to (x) > (y); however, unlike (x) > (y), isgreater(x, y) shall not raise the invalid floating-point exception when x and y are unordered.

RETURN VALUE

Upon successful completion, the isgreater() macro shall return the value of (x) > (y).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isgreaterequal(), isless(), islessequal(), islessgreater(), isunordered(), the Base Definitions volume of IEEE Std 1003.1-2001 <math.h>

CHANGE HISTORY

isgreaterequal() — test if x is greater than or equal to y

#include <math.h>

int isgreaterequal(real-floating x, real-floating y);

The isgreaterequal() macro shall determine whether its first argument is greater than or equal to its second argument. The value of isgreaterequal(x, y) shall be equal to (x) >= (y), however, unlike (x) >= (y), isgreaterequal(x, y) shall not raise the invalid floating-point exception when x and y are unordered.

Upon successful completion, the isgreaterequal() macro shall return the value of (x) >= (y).

If x or y is NaN, 0 shall be returned.

No errors are defined.

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.

No errors are defined.

isgreater(), isless(), islessequal(), islessgreater(), isunordered(), the Base Definitions volume of IEEE Std 1003.1-2001 <math.h>

NAME
  isinf — test for infinity

SYNOPSIS
  #include <math.h>
  int isinf(real-floating x);

DESCRIPTION
  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
  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.

RETURN VALUE
  The isinf() macro shall return a non-zero value if and only if its argument has an infinite value.

ERRORS
  No errors are defined.

EXAMPLES
  None.

APPLICATION USAGE
  None.

RATIONALE
  None.

FUTURE DIRECTIONS
  None.

SEE ALSO
  fpclassify(), isfinite(), isnan(), isnormal(), signbit(), the Base Definitions volume of
  IEEE Std 1003.1-2001 <math.h>

CHANGE HISTORY
NAME
isless — test if x is less than y

SYNOPSIS
#include <math.h>

int isless(real-floating x, real-floating y);

DESCRIPTION
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 isless() macro shall determine whether its first argument is less than its second argument.

The value of isless(x, y) shall be equal to (x) < (y); however, unlike (x) < (y), isless(x, y) shall not raise the invalid floating-point exception when x and y are unordered.

RETURN VALUE
Upon successful completion, the isless() macro shall return the value of (x) < (y).

If x or y is NaN, 0 shall be returned.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
isgreater(), isgreaterequal(), islessequal(), islessgreater(), isnunordered(), the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

CHANGE HISTORY
islessequal()  

NAME  
islessequal — test if x is less than or equal to y  

SYNOPSIS  
#include <math.h>  
int islessequal(real-floating x, real-floating y);  

DESCRIPTION  
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.  

The islessequal() macro shall determine whether its first argument is less than or equal to its second argument. The value of islessequal(x, y) shall be equal to (x) <= (y); however, unlike (x) <= (y), islessequal(x, y) shall not raise the invalid floating-point exception when x and y are unordered.  

RETURN VALUE  
Upon successful completion, the islessequal() macro shall return the value of (x) <= (y).  

If x or y is NaN, 0 shall be returned.  

ERRORS  
No errors are defined.  

EXAMPLES  
None.  

APPLICATION USAGE  
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.  

RATIONALE  
None.  

FUTURE DIRECTIONS  
None.  

SEE ALSO  
igreater(), isgreaterequal(), isless(), islessgreater(), isunordered(), the Base Definitions volume of IEEE Std 1003.1-2001 <math.h>  

CHANGE HISTORY  
NAME

islessgreater — test if x is less than or greater than y

SYNOPSIS

#include <math.h>

int islessgreater(real-floating x, real-floating y);

DESCRIPTION

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

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) || (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).

RETURN VALUE

Upon successful completion, the islessgreater() macro shall return the value of
(x) < (y) || (x) > (y).

If x or y is NaN, 0 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isgreater(), isgreaterequal(), isless(), islessequal(), isunordered(), the Base Definitions volume of
IEEE Std 1003.1-2001 <math.h>

CHANGE HISTORY

NAME
islower — test for a lowercase letter

SYNOPSIS
#include <ctype.h>
int islower(int c);

DESCRIPTION
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 `islower()` function shall test whether `c` is 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.

The `c` argument is an `int`, the value of which the application shall ensure is a character 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.

RETURN VALUE
The `islower()` function shall return non-zero if `c` is a lowercase letter; otherwise, it shall return 0.

ERRORS
No errors are defined.

EXAMPLES
Testing for a Lowercase Letter
The following example tests whether the value is a lowercase letter, based on the locale of the user, then uses it as part of a key value.

```c
#include <ctype.h>
#include <stdlib.h>
#include <locale.h>
...
char *keystr;
int elementlen, len;
char c;
...
setlocale(LC_ALL, "");
...
len = 0;
while (len < elementlen) {
    c = (char) (rand() % 256);
    ...
   if (islower(c))
       keystr[len++] = c;
...
```

APPLICATION USAGE
To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.
**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, `<ctype.h>`

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 6**
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
An example is added.
isnan()

NAME
isnan — test for a NaN

SYNOPSIS
#include <math.h>
int isnan(real-floating x);

DESCRIPTION
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

The isnan( ) macro shall determine whether its argument value is a NaN. 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.

RETURN VALUE
The isnan( ) macro shall return a non-zero value if and only if its argument has a NaN value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fclassify(), isfinite(), isinf(), isnormal(), signbit(), the Base Definitions volume of
IEEE Std 1003.1-2001, <math.h>

CHANGE HISTORY
First released in Issue 3.

Issue 5
The DESCRIPTION is updated to indicate the return value when NaN is not supported. This
text was previously published in the APPLICATION USAGE section.

Issue 6
Entry re-written for alignment with the ISO/IEC 9899:1999 standard.
NAME

isnormal — test for a normal value

SYNOPSIS

#include <math.h>

int isnormal(real-floating x);

DESCRIPTION

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 isnormal() macro shall determine whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). 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.

RETURN VALUE

The isnormal() macro shall return a non-zero value if and only if its argument has a normal value.

ERRORS

No errors are defined.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fpclassify(), isfinite(), isninf(), isnan(), signbit(), the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

CHANGE HISTORY

NAME
isprint — test for a printable character

SYNOPSIS
#include <ctype.h>
int isprint(int c);

DESCRIPTION
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

The isprint() function shall test whether c is 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.

The c argument is an int, the value of which the application shall ensure is a character
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.

RETURN VALUE
The isprint() function shall return non-zero if c is a printable character; otherwise, it shall return
0.

ERRORS
No errors are defined.

APPLICATION USAGE
To ensure applications portability, especially across natural languages, only this function and
those listed in the SEE ALSO section should be used for character classification.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), ispunct(), isspace(), isupper(),
isxdigit(), setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale,
<ctype.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
ispunct — test for a punctuation character

SYNOPSIS
#include <ctype.h>
int ispunct(int c);

DESCRIPTION
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

The ispunct() function shall test whether c is a character of class punct in the program’s current
locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

The c argument is an int, the value of which the application shall ensure is a character
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.

RETURN VALUE
The ispunct() function shall return non-zero if c is a punctuation character; otherwise, it shall
return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
To ensure applications portability, especially across natural languages, only this function and
those listed in the SEE ALSO section should be used for character classification.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), isspace(), isupper(), isxdigit(),
setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
issepace() — test for a white-space character

#include <ctype.h>

int isspace(int c);

DESCRIPTION
The isspace() function shall test whether c is a character of class space in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

The c argument is an int, the value of which the application shall ensure is a character 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.

RETURN VALUE
The isspace() function shall return non-zero if c is a white-space character; otherwise, it shall return 0.

ERRORS
No errors are defined.

APPLICATION USAGE
To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isupper(), isxdigit(), setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
isunordered — test if arguments are unordered

SYNOPSIS
#include <math.h>
int isunordered(real-floating x, real-floating y);

DESCRIPTION
The functionality described on this reference page is aligned with the ISO C standard. Any
collision between the requirements described here and the ISO C standard is unintentional. This
The isunordered() macro shall determine whether its arguments are unordered.

RETURN VALUE
Upon successful completion, the isunordered() macro shall return 1 if its arguments are
unordered, and 0 otherwise.
If x or y is NaN, 0 shall be returned.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
isgreater(), isgreaterequal(), isless(), islessequal(), islessgreater(), the Base Definitions volume of
IEEE Std 1003.1-2001, <math.h>

CHANGE HISTORY
isupper()  

NAME  
isupper — test for an uppercase letter

SYNOPSIS  
#include <ctype.h>

int isupper(int c);

DESCRIPTION  
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 isupper() function shall test whether c is 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.

The c argument is an int, the value of which the application shall ensure is a character 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.

RETURN VALUE  
The isupper() function shall return non-zero if c is an uppercase letter; otherwise, it shall return 0.

ERRORS  
No errors are defined.

EXAMPLES  
None.

APPLICATION USAGE  
To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE  
None.

FUTURE DIRECTIONS  
None.

SEE ALSO  
isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isxdigit(), setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

CHANGE HISTORY  
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6  
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME

iswalnum — test for an alphanumeric wide-character code

SYNOPSIS

#include <wctype.h>

int iswalnum(wint_t wc);

DESCRIPTION

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 iswalnum() function shall test whether wc is a wide-character code representing a character of class alpha or digit in the program’s current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

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.

RETURN VALUE

The iswalnum() function shall return non-zero if wc is an alphanumeric wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalpha(), iswcntrl(), iswctype(), iswdigit(), iswgraph(), iswlower(), iswprint(), iswpunct(), iswspace(), iswupper(), iswxdigit(), setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <stdio.h>, <wchar.h>, <wctype.h>

CHANGE HISTORY

First released as a World-wide Portability Interface in Issue 4.

Issue 5

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>.

Issue 6

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
iswalpha() — test for an alphabetic wide-character code

**SYNOPSIS**

```c
#include <wctype.h>

int iswalpha(wint_t wc);
```

**DESCRIPTION**

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 *iswalpha()* function shall test whether `wc` is a wide-character code representing 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.

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.

**RETURN VALUE**

The *iswalpha()* function shall return non-zero if `wc` is an alphabetic wide-character code; otherwise, it shall return 0.

**ERRORS**

No errors are defined.

**EXAMPLES**

None.

**APPLICATION USAGE**

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.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`iswalnum()`, `iswcntrl()`, `iswctype()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, `<stdio.h>`, `<wchar.h>`, `<wctype.h>`

**CHANGE HISTORY**

First released in Issue 4.

**Issue 5**

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>`.

**Issue 6**

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
iswblank — test for a blank wide-character code

SYNOPSIS
#include <wctype.h>

int iswblank(wint_t wc);

DESCRIPTION
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

The iswblank() function shall test whether wc is a wide-character code representing a character of
class blank in the program’s current locale; see the Base Definitions volume of

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.

RETURN VALUE
The iswblank() function shall return non-zero if wc is a blank wide-character code; otherwise, it
shall return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
iswalnum(), iswalpha(), iswcntrl(), iswctype(), iswdigit(), iswgraph(), iswlower(), iswprint(),
isupunct(), iswspace(), iswupper(), iswxdigit(), setlocale(), the Base Definitions volume of
IEEE Std 1003.1-2001, Chapter 7, Locale, <stdio.h>, <wchar.h>, <wctype.h>

CHANGE HISTORY
NAME
iswcntrl — test for a control wide-character code

SYNOPSIS
#include <wctype.h>

int iswcntrl(wint_t wc);

DESCRIPTION
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.

The iswcntrl() function shall test whether wc is a wide-character code representing a character of class cntrl in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

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.

RETURN VALUE
The iswcntrl() function shall return non-zero if wc is a control wide-character code; otherwise, it shall return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
iswalnum(), iswalpha(), iswctype(), iswxdigit(), iswgraph(), iswalower(), iswprint(), iswpunct(), iswspace(), iswupper(), iswxdigit(), setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

CHANGE HISTORY
First released in Issue 4.

Issue 5
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>.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
iswctype — test character for a specified class

SYNOPSIS
#include <wctype.h>

int iswctype(wint_t wc, wctype_t charclass);

DESCRIPTION
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 iswctype() function shall determine whether the wide-character code wc has the character class charclass, returning true or false. The iswctype() function is defined on WEOF and wide-character codes corresponding to the valid character encodings in the current locale. If the wc argument is not in the domain of the function, the result is undefined. If the value of charclass is invalid (that is, not obtained by a call to wctype() or charclass is invalidated by a subsequent call to setlocale() that has affected category LC_CTYPE) the result is unspecified.

RETURN VALUE
The iswctype() function shall return non-zero (true) if and only if wc has the property described by charclass. If charclass is 0, iswctype() shall return 0.

ERRORS
No errors are defined.

EXAMPLES
Testing for a Valid Character
#include <wctype.h>
...
int yes_or_no;
wint_t wc;
wctype_t valid_class;
...
if ((valid_class=wctype("vowel")) == (wctype_t)0)
   /* Invalid character class. */
yes_or_no=iswctype(wc,valid_class);

APPLICATION USAGE
The twelve strings "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper", and "xdigit" are reserved for the standard character classes. In the table below, the functions in the left column are equivalent to the functions in the right column.
iswalnum(wc)  iswctype(wc, wctype("alnum"))
iswalpha(wc)  iswctype(wc, wctype("alpha"))
iswblank(wc)  iswctype(wc, wctype("blank"))
iswcntrl(wc)  iswctype(wc, wctype("cntrl"))
iswctrl(wc)  iswctype(wc, wctype("ctrl"))
iswgraph(wc)  iswctype(wc, wctype("graph"))
iswlower(wc)  iswctype(wc, wctype("lower"))
iswprint(wc)  iswctype(wc, wctype("print"))
iswpunct(wc)  iswctype(wc, wctype("punct"))
iswspace(wc)  iswctype(wc, wctype("space"))
iswctype()

System Interfaces

iswupper(wc) iswctype(wc, wctype("upper"))

iswxdigit(wc) iswctype(wc, wctype("xdigit"))

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
iswalnum(), iswalpha(), iswcntrl(), iswdigit(), iswgraph(), iswlower(), iswprint(), iswpunct(), iswspace(), iswupper(), iswxdigit(), setlocale(), wctype(), the Base Definitions volume of
IEEE Std 1003.1-2001, <wchar.h>, <wctype.h>

CHANGE HISTORY
First released as World-wide Portability Interfaces in Issue 4.

Issue 5
The following change has been made in this issue for alignment with
- 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>.

Issue 6
The behavior of n=0 is now described.
An example is added.
A new function, iswblank(), is added to the list in the APPLICATION USAGE.
NAME
iswdigit — test for a decimal digit wide-character code

SYNOPSIS
#include <wctype.h>

int iswdigit(wint_t wc);

DESCRIPTION
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 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.

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.

RETURN VALUE
The iswdigit() function shall return non-zero if wc is a decimal digit wide-character code; otherwise, it shall return 0.

ERRORS
No errors are defined.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
iswalnum(), iswalpha(), iswcntrl(), iswctype(), iswgraph(), iswlower(), iswprint(), iswpunct(), iswspace(), iswupper(), iswxdigit(), setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

CHANGE HISTORY
First released in Issue 4.

Issue 5
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>.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME

iswgraph — test for a visible wide-character code

SYNOPSIS
#include <wctype.h>

int iswgraph(wint_t wc);

DESCRIPTION

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

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

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.

RETURN VALUE

The iswgraph() function shall return non-zero if wc is a wide-character code with a visible
representation; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum(), iswalpha(), iswcntrl(), iswctype(), iswdigit(), iswlower(), iswprint(), iswpunct(),
iswspace(), iswupper(), iswxdigit(), setlocale(), the Base Definitions volume of
IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with

• 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>.

Issue 6

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

iswlower — test for a lowercase letter wide-character code

SYNOPSIS

#include <wctype.h>

int iswlower(wint_t wc);

DESCRIPTION

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 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.

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.

RETURN VALUE

The iswlower() function shall return non-zero if wc is a lowercase letter wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

APPLICATION USAGE

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum(), iswalpha(), iswcntrl(), iswctype(), iswdigit(), iswgraph(), iswprint(), iswpunct(), iswspace(), iswupper(), iswxdigit(), setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

CHANGE HISTORY

First released in Issue 4.

Issue 5
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>.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME

iswprint — test for a printable wide-character code

SYNOPSIS

#include <wctype.h>

int iswprint(wint_t wc);

DESCRIPTION

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.

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.

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.

RETURN VALUE

The iswprint() function shall return non-zero if wc is a printable wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum(), iswalph(), iswcntrl(), iswctype(), iswdigit(), iswgraph(), iswlower(), iswpunct(), iswspace(), iswupper(), iswxdigit(), setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

CHANGE HISTORY

Issue 5

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>.

Issue 6

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
iswpunct — test for a punctuation wide-character code

SYNOPSIS
#include <wctype.h>

int iswpunct(wint_t wc);

DESCRIPTION
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

The iswpunct() function shall test whether wc is a wide-character code representing a character
of class punct in the program’s current locale; see the Base Definitions volume of

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.

RETURN VALUE
The iswpunct() function shall return non-zero if wc is a punctuation wide-character code;
otherwise, it shall return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
iswalnum(), iswalnum(), iswctype(), iswdigit(), iswgraph(), iswlower(), iswprint(),
iswspace(), iswupper(), iswxdigit(), setlocale(), the Base Definitions volume of
IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

CHANGE HISTORY
First released in Issue 4.

Issue 5
The following change has been made in this issue for alignment with
• 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>.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
iswspace — test for a white-space wide-character code

SYNOPSIS
#include <wctype.h>

int iswspace(wint_t wc);

DESCRIPTION
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

The iswspace() function shall test whether wc is a wide-character code representing a character of
class space in the program’s current locale; see the Base Definitions volume of

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.

RETURN VALUE
The iswspace() function shall return non-zero if wc is a white-space wide-character code;
otherwise, it shall return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
iswalnum(), iswalpha(), iswcntrl(), iswctype(), iswdigit(), iswgraph(), iswlower(), iswprint(),
iswpunct(), iswupper(), iswxdigit(), setlocale(), the Base Definitions volume of
IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

CHANGE HISTORY
First released in Issue 4.

Issue 5
The following change has been made in this issue for alignment with

• 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>.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
iswupper — test for an uppercase letter wide-character code

SYNOPSIS
#include <wctype.h>

int iswupper(wint_t wc);

DESCRIPTION
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

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

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.

RETURN VALUE
The iswupper() function shall return non-zero if wc is an uppercase letter wide-character code;
otherwise, it shall return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
iswalnum(), iswalpha(), iswcntrl(), iswctype(), iswdigit(), iswgraph(), iswlower(), iswprint(),
iswpunct(), iswspace(), iswxdigit(), setlocale(), the Base Definitions volume of
IEEE Std 1003.1-2001, Chapter 7, Locale, <wchar.h>, <wctype.h>

CHANGE HISTORY
First released in Issue 4.

Issue 5
The following change has been made in this issue for alignment with

• 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>.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
iswxdigit() — test for a hexadecimal digit wide-character code

```
#include <wctype.h>

int iswxdigit(wint_t wc);
```

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.

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.

The `iswxdigit()` function shall return non-zero if `wc` is a hexadecimal digit wide-character code; otherwise, it shall return 0.

No errors are defined.

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.

IEEE Std 1003.1-2001, Chapter 7, Locale, `<wchar.h>`, `<wctype.h>`

First released in Issue 4.

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>`.

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

Issue 5

Issue 6
NAME
isdigit — test for a hexadecimal digit

SYNOPSIS
#include <ctype.h>
int isxdigit(int c);

DESCRIPTION
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 isxdigit() function shall test whether \( c \) is a character of class \( \text{xdigit} \) in the program’s current locale; see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale.

The \( c \) argument is an int, the value of which the application shall ensure is a character 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.

RETURN VALUE
The isxdigit() function shall return non-zero if \( c \) is a hexadecimal digit; otherwise, it shall return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(),
the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
j0, j1, jn — Bessel functions of the first kind

SYNOPSIS
#include <math.h>

double j0(double x);
double j1(double x);
double jn(int n, double x);

DESCRIPTION
The j0(), j1(), and jn() functions shall compute Bessel functions of x of the first kind of orders 0, 1, and n, respectively.

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.

RETURN VALUE
Upon successful completion, these functions shall return the relevant Bessel value of x of the first kind.

If the x argument is too large in magnitude, or the correct result would cause underflow, 0 shall be returned and a range error may occur.

If x is NaN, a NaN shall be returned.

ERRORS
These functions may fail if:

Range Error: The value of x was too large in magnitude, or an underflow occurred.

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.

No other errors shall occur.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fearlearexcept(), fetestexcept(), isnan(), y0(), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>
CHANGE HISTORY

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

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The may fail [EDOM] error is removed for the case for NaN.
The RETURN VALUE and ERRORS sections are reworked for alignment of the error handling with the ISO/IEC 9899: 1999 standard.
NAME
jrand48 — generate a uniformly distributed pseudo-random long signed integer

SYNOPSIS
#include <stdlib.h>

long jrand48(unsigned short xsubi[3]);

DESCRIPTION
Refer to drand48().
NAME
kill — send a signal to a process or a group of processes

SYNOPSIS
#include <signal.h>

int kill(pid_t pid, int sig);

DESCRIPTION
The kill() function shall send a signal to a process or a group of processes specified by pid. The signal to be sent is specified by sig and is either one from the list given in <signal.h> or 0. If sig is 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of pid.

For a process to have permission to send a signal to a process designated by pid, unless the sending process has appropriate privileges, the real or effective user ID of the sending process shall match the real or saved set-user-ID of the receiving process.

If pid is greater than 0, sig shall be sent to the process whose process ID is equal to pid.

If pid is 0, sig shall be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the process group ID of the sender, and for which the process has permission to send a signal.

If pid is −1, sig shall be sent to all processes (excluding an unspecified set of system processes) for which the process has permission to send that signal.

If pid is negative, but not −1, sig shall be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the absolute value of pid, and for which the process has permission to send a signal.

If the value of pid causes sig to be generated for the sending process, and if sig is not blocked for the calling thread and no other thread has sig unblocked or is waiting in a sigwait() function for sig, either sig or at least one pending unblocked signal shall be delivered to the sending thread before kill() returns.

The user ID tests described above shall not be applied when sending SIGCONT to a process that is a member of the same session as the sending process.

An implementation that provides extended security controls may impose further implementation-defined restrictions on the sending of signals, including the null signal. In particular, the system may deny the existence of some or all of the processes specified by pid.

The kill() function is successful if the process has permission to send sig to any of the processes specified by pid. If kill() fails, no signal shall be sent.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to indicate the error.

ERRORS
The kill() function shall fail if:

EINVAL The value of the sig argument is an invalid or unsupported signal number.

EPERM The process does not have permission to send the signal to any receiving process.

ESRCH No process or process group can be found corresponding to that specified by pid.

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The semantics for permission checking for kill() differed between System V and most other implementations, such as Version 7 or 4.3 BSD. The semantics chosen for this volume of IEEE Std 1003.1-2001 agree with System V. Specifically, a set-user-ID process cannot protect itself against signals (or at least not against SIGKILL) unless it changes its real user ID. This choice allows the user who starts an application to send it signals even if it changes its effective user ID. The other semantics give more power to an application that wants to protect itself from the user who ran it.

Some implementations provide semantic extensions to the kill() function when the absolute value of pid is greater than some maximum, or otherwise special, value. Negative values are a flag to kill(). Since most implementations return [ESRCH] in this case, this behavior is not included in this volume of IEEE Std 1003.1-2001, although a conforming implementation could provide such an extension.

The implementation-defined processes to which a signal cannot be sent may include the scheduler or init.

There was initially strong sentiment to specify that, if pid specifies that a signal be sent to the calling process and that signal is not blocked, that signal would be delivered before kill() returns. This would permit a process to call kill() and be guaranteed that the call never return. However, historical implementations that provide only the signal() function make only the weaker guarantee in this volume of IEEE Std 1003.1-2001, because they only deliver one signal each time a process enters the kernel. Modifications to such implementations to support the sigaction() function generally require entry to the kernel following return from a signal-catching function, in order to restore the signal mask. Such modifications have the effect of satisfying the stronger requirement, at least when sigaction() is used, but not necessarily when signal() is used.

The developers of this volume of IEEE Std 1003.1-2001 considered making the stronger requirement except when signal() is used, but felt this would be unnecessarily complex. Implementors are encouraged to meet the stronger requirement whenever possible. In practice, the weaker requirement is the same, except in the rare case when two signals arrive during a very short window. This reasoning also applies to a similar requirement for sigprocmask().

In 4.2 BSD, the SIGCONT signal can be sent to any descendant process regardless of user-ID security checks. This allows a job control shell to continue a job even if processes in the job have altered their user IDs (as in the su command). In keeping with the addition of the concept of sessions, similar functionality is provided by allowing the SIGCONT signal to be sent to any process in the same session regardless of user ID security checks. This is less restrictive than BSD in the sense that ancestor processes (in the same session) can now be the recipient. It is more restrictive than BSD in the sense that descendant processes that form new sessions are now subject to the user ID checks. A similar relaxation of security is not necessary for the other job control signals since those signals are typically sent by the terminal driver in recognition of special characters being typed; the terminal driver bypasses all security checks.

In secure implementations, a process may be restricted from sending a signal to a process having a different security label. In order to prevent the existence or nonexistence of a process from being used as a covert channel, such processes should appear nonexistent to the sender; that is, [ESRCH] should be returned, rather than [EPERM], if pid refers only to such processes.
Existing implementations vary on the result of a `kill()` with `pid` indicating an inactive process (a terminated process that has not been waited for by its parent). Some indicate success on such a call (subject to permission checking), while others give an error of [ESRCH]. Since the definition of process lifetime in this volume of IEEE Std 1003.1-2001 covers inactive processes, the [ESRCH] error as described is inappropriate in this case. In particular, this means that an application cannot have a parent process check for termination of a particular child with `kill()`. (Usually this is done with the null signal; this can be done reliably with `waitpid()`.)

There is some belief that the name `kill()` is misleading, since the function is not always intended to cause process termination. However, the name is common to all historical implementations, and any change would be in conflict with the goal of minimal changes to existing application code.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`getpid()`, `raise()`, `setsid()`, `sigaction()`, `sigqueue()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`, `<sys/types.h>`

**CHANGE HISTORY**

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

**Issue 5**

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

**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:

- In the DESCRIPTION, the second paragraph is reworded to indicate that the saved set-user-ID of the calling process is checked in place of its effective user ID. This is a FIPS requirement.

- 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.

- The behavior when `pid` is −1 is now specified. It was previously explicitly unspecified in the POSIX.1-1988 standard.

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

killpg — send a signal to a process group

**SYNOPSIS**

```c
#include <signal.h>

int killpg(pid_t pgrp, int sig);
```

**DESCRIPTION**

The `killpg()` function shall send the signal specified by `sig` to the process group specified by `pgrp`.

If `pgrp` is greater than 1, `killpg(pgrp, sig)` shall be equivalent to `kill(−pgrp, sig)`. If `pgrp` is less than or equal to 1, the behavior of `killpg()` is undefined.

**RETURN VALUE**

Refer to `kill()`.

**ERRORS**

Refer to `kill()`.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`getpgid()`, `getpid()`, `kill()`, `raise()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`

**CHANGE HISTORY**

First released in Issue 4, Version 2.

**Issue 5**

Moved from X/OPEN UNIX extension to BASE.
NAME
l64a — convert a 32-bit integer to a radix-64 ASCII string

SYNOPSIS

```c
#include <stdlib.h>

char *l64a(long value);
```

DESCRIPTION
Refer to a64l().
NAME
labs, llabs — return a long integer absolute value

SYNOPSIS
#include <stdlib.h>
long labs(long i);
long long llabs(long long i);

DESCRIPTION
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 labs() function shall compute the absolute value of the long integer operand i. The llabs() function shall compute the absolute value of the long long integer operand i. If the result cannot be represented, the behavior is undefined.

RETURN VALUE
The labs() function shall return the absolute value of the long integer operand. The labs() function shall return the absolute value of the long long integer operand.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
abs(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY
First released in Issue 4. Derived from the ISO C standard.

Issue 6
The llabs() function is added for alignment with the ISO/IEC 9899:1999 standard.
NAME
Ichown — change the owner and group of a symbolic link

SYNOPSIS
#include <unistd.h>

int lchown(const char *path, uid_t owner, gid_t group);

DESCRIPTION
The lchown() function shall be equivalent to chown(), except in the case where the named file is a symbolic link. In this case, lchown() shall change the ownership of the symbolic link file itself, while chown() changes the ownership of the file or directory to which the symbolic link refers.

RETURN VALUE
Upon successful completion, lchown() shall return 0. Otherwise, it shall return −1 and set errno to indicate an error.

ERRORS
The lchown() function shall fail if:

[EACCES] Search permission is denied on a component of the path prefix of path.

EINVAL] The owner or group ID is not a value supported by the implementation.

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

[ENAMETOOLONG] The length of a pathname exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

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

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

[EOPNOTSUPP] The path argument names a symbolic link and the implementation does not support setting the owner or group of a symbolic link.

[EPERM] The effective user ID does not match the owner of the file and the process does not have appropriate privileges.

[EIO] An I/O error occurred while reading or writing to the file system.

[EINTR] A signal was caught during execution of the function.

[ELOOP] More than SYMLOOP_MAX symbolic links were encountered during resolution of the path argument.

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
EXAMPLES

Changing the Current Owner of a File

The following example shows how to change the ownership of the symbolic link named /modules/pass1 to the user ID associated with “jones” and the group ID associated with “cnd”.

The numeric value for the user ID is obtained by using the getpwnam() function. The numeric value for the group ID is obtained by using the getgrnam() function.

```c
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>

struct passwd *pwd;
struct group *grp;
char *path = "/modules/pass1";
...
pwd = getpwnam("jones");
grp = getgrnam("cnd");
lchown(path, pwd->pw_uid, grp->gr_gid);
```

APPLICATION USAGE

On implementations which support symbolic links as directory entries rather than files, lchown() may fail.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chown(), symlink(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

The Open Group Base Resolution bwg2001-013 is applied, adding wording to the APPLICATION USAGE.
NAME
lcong48 — seed a uniformly distributed pseudo-random signed long integer generator

SYNOPSIS
XSI
#include <stdlib.h>

void lcong48(unsigned short param[7]);

DESCRIPTION
Refer to drand48().
**NAME**
ldexp, ldexpf, ldexpl — load exponent of a floating-point number

**SYNOPSIS**
```c
#include <math.h>

double ldexp(double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
```

**DESCRIPTION**
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.

These functions shall compute the quantity \( x \times 2^\text{exp} \).

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.

**RETURN VALUE**
Upon successful completion, these functions shall return \( x \) multiplied by \( 2 \), raised to the power \( \text{exp} \).

If these functions would cause overflow, a range error shall occur and `ldexp()`, `ldexpf()`, and `ldexpl()` shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (according to the sign of \( x \)), respectively.

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.

If \( x \) is NaN, a NaN shall be returned.

If \( x \) is ±0 or ±Inf, \( x \) shall be returned.

If \( \text{exp} \) is 0, \( x \) shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

**ERRORS**
These functions shall fail if:

**Range Error**
The result overflows.

If the integer expression `(math_errno & MATH_ERRNO)` is non-zero, then `errno` shall be set to `[ERANGE]`. If the integer expression `(math_errno & MATH_ERREXCEPT)` is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

**Range Error**
The result underflows.

If the integer expression `(math_errno & MATH_ERRNO)` is non-zero, then `errno` shall be set to `[ERANGE]`. If the integer expression `(math_errno & MATH_ERREXCEPT)` is non-zero, then the underflow floating-point exception shall be raised.
EXAMPLES
None.

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

RATIONALE
None.

FUTURE DIRECTIONS
None.

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

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The ldexpf() and ldexpl() functions are added for alignment with the ISO/IEC 9899:1999 standard.

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

NAME
ldiv, lldiv — compute quotient and remainder of a long division

SYNOPSIS
#include <stdlib.h>

ldiv_t ldiv(long numer, long denom);
lldiv_t lldiv(long long numer, long long denom);

DESCRIPTION
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

These functions shall compute the quotient and remainder of the division of the numerator
numer by the denominator denom. If the division is inexact, the resulting quotient is the long
integer (for the ldiv() function) or long long integer (for the lldiv() function) of lesser magnitude
that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is
undefined; otherwise, quot * denom + rem shall equal numer.

RETURN VALUE
The ldiv() function shall return a structure of type ldiv_t, comprising both the quotient and the
remainder. The structure shall include the following members, in any order:

long quot; /* Quotient */
long rem;  /* Remainder */

The lldiv() function shall return a structure of type lldiv_t, comprising both the quotient and the
remainder. The structure shall include the following members, in any order:

long long quot; /* Quotient */
long long rem;  /* Remainder */

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
div(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY
First released in Issue 4. Derived from the ISO C standard.

Issue 6
The lldiv() function is added for alignment with the ISO/IEC 9899:1999 standard.
NAME
Ifind — find entry in a linear search table

SYNOPSIS
#include <search.h>

void *lfind(const void *key, const void *base, size_t *nelp,
    size_t width, int (*compar)(const void *, const void *));

DESCRIPTION
Refer to lsearch().
NAME
lgamma, lgammaf, lgammal — log gamma function

SYNOPSIS
#include <math.h>

double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
extern int signgam;

DESCRIPTION
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

These functions shall compute \( \log \Gamma(x) \) where \( \Gamma(x) \) is defined as
\[
\int_0^\infty e^{-t} t^{x-1} \, dt.
\]
The argument \( x \) need not be a non-positive integer (\( \Gamma(x) \) is defined over the reals, except the non-positive
integers).

The sign of \( \Gamma(x) \) is returned in the external integer \texttt{signgam}.

These functions need not be reentrant. A function that is not required to be reentrant is not
required to be thread-safe.

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

RETURN VALUE
Upon successful completion, these functions shall return the logarithmic gamma of \( x \).

If \( x \) is a non-positive integer, a pole error shall occur and \texttt{lgamma()}, \texttt{lgammaf()}, and \texttt{lgammal()} shall return \(+\texttt{HUGE_VAL} \), \(+\texttt{HUGE_VALF} \), and \(+\texttt{HUGE_VALL} \), respectively.

If the correct value would cause overflow, a range error shall occur and \texttt{lgamma()}, \texttt{lgammaf()},
and \texttt{lgammal()} shall return \(\pm\texttt{HUGE_VAL} \), \(\pm\texttt{HUGE_VALF} \), and \(\pm\texttt{HUGE_VALL} \) (having the same
sign as the correct value), respectively.

If \( x \) is NaN, a NaN shall be returned.

If \( x \) is 1 or 2, +0 shall be returned.

If \( x \) is \(\pm\texttt{Inf} \), +Inf shall be returned.

ERRORS
These functions shall fail if:

Pole Error \( x \) argument is a negative integer or zero.

If the integer expression (\texttt{math_errhandling} & \texttt{MATH_ERRNO}) is non-zero,
then \texttt{errno} shall be set to \[\text{ERANGE} \]. If the integer expression
\texttt{math_errhandling} & \texttt{MATH_ERREXCEPT} is non-zero, then the divide-by-
zero floating-point exception shall be raised.

Range Error The result overflows.
If the integer expression (math_errno == MATH_ERRNO) is non-zero,
then errno shall be set to [ERANGE]. If the integer expression
(math_errno == MATH_ERREXCEPT) is non-zero, then the overflow
floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

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

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exp(), 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>

CHANGE HISTORY

First released in Issue 3.

Issue 5

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

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

The lgamma() function is no longer marked as an extension.

The lgammaf() and lgammal() functions are added for alignment with the ISO/IEC 9899:1999
standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
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.

XSI extensions are marked.
NAME
link — link to a file

SYNOPSIS
#include <unistd.h>

int link(const char *path1, const char *path2);

DESCRIPTION
The link() function shall create a new link (directory entry) for the existing file, path1.
The path1 argument points to a pathname naming an existing file. The path2 argument points to
a pathname naming the new directory entry to be created. The link() function shall atomically
create a new link for the existing file and the link count of the file shall be incremented by one.
If path1 names a directory, link() shall fail unless the process has appropriate privileges and the
implementation supports using link() on directories.
Upon successful completion, link() shall mark for update the st_ctime field of the file. Also, the
st_ctime and st_mtime fields of the directory that contains the new entry shall be marked for
update.
If link() fails, no link shall be created and the link count of the file shall remain unchanged.
The implementation may require that the calling process has permission to access the existing
file.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to
indicate the error.

ERRORS
The link() function shall fail if:

[EACCES] A component of either path prefix denies search permission, or the requested
link requires writing in a directory that denies write permission, or the calling
process does not have permission to access the existing file and this is
required by the implementation.

[EXIST] The path2 argument resolves to an existing file or refers to a symbolic link.

[ELOOP] A loop exists in symbolic links encountered during resolution of the path1 or
path2 argument.

[ENL] The number of links to the file named by path1 would exceed [LINK_MAX].

[ENAMEETOOLONG] The length of the path1 or path2 argument exceeds [PATH_MAX] or a
pathname component is longer than [NAME_MAX].

[ENOENT] A component of either path prefix does not exist; the file named by path1 does
not exist; or path1 or path2 points to an empty string.

[ENOSPC] The directory to contain the link cannot be extended.

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

[EPERM] The file named by path1 is a directory and either the calling process does not
have appropriate privileges or the implementation prohibits using link() on
directories.
The requested link requires writing in a directory on a read-only file system.

The link named by path2 and the file named by path1 are on different file systems and the implementation does not support links between file systems.

path1 refers to a named STREAM.

The link() function may fail if:

More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the path1 or path2 argument.

As a result of encountering a symbolic link in resolution of the path1 or path2 argument, the length of the substituted pathname string exceeded {PATH_MAX}.

**EXAMPLES**

**Creating a Link to a File**

The following example shows how to create a link to a file named /home/cnd/mod1 by creating a new directory entry named /modules/pass1.

```c
#include <unistd.h>

char *path1 = "/home/cnd/mod1";
char *path2 = "/modules/pass1";
int status;
...
status = link(path1, path2);
```

**Creating a Link to a File Within a Program**

In the following program example, the link() function links the /etc/passwd file (defined as PASSWDFILE) to a file named /etc/opasswd (defined as SAVEFILE), which is used to save the current password file. Then, after removing the current password file (defined as PASSWDFILE), the new password file is saved as the current password file using the link() function again.

```c
#include <unistd.h>

#define LOCKFILE "/etc/ptmp"
define PASSWDFILE "/etc/passwd"
define SAVEFILE "/etc/opasswd"
...
/* Save current password file */
link(PASSWDFILE, SAVEFILE);
/* Remove current password file. */
unlink(PASSWDFILE);
/* Save new password file as current password file. */
link(LOCKFILE, PASSWDFILE);
```
APPLICATION USAGE

Some implementations do allow links between file systems.

RATIONALE

Linking to a directory is restricted to the superuser in most historical implementations because this capability may produce loops in the file hierarchy or otherwise corrupt the file system. This volume of IEEE Std 1003.1-2001 continues that philosophy by prohibiting `link()` and `unlink()` from doing this. Other functions could do it if the implementor designed such an extension.

Some historical implementations allow linking of files on different file systems. Wording was added to explicitly allow this optional behavior.

The exception for cross-file system links is intended to apply only to links that are programmatically indistinguishable from “hard” links.

FUTURE DIRECTIONS

None.

SEE ALSO

`symlink()`, `unlink()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<unistd.h>`

CHANGE HISTORY

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

Issue 6

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

- The [ELOOP] mandatory error condition is added.

- A second [ENAMETOOLONG] is added as an optional error condition.

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

- An explanation is added of the action when `path2` refers to a symbolic link.

- The [ELOOP] optional error condition is added.
NAME
lio_listio — list directed I/O (REALTIME)

SYNOPSIS
#include <aio.h>

int lio_listio(int mode, struct aiocb *restrict const list[restrict],
int nent, struct sigevent *restrict sig);

DESCRIPTION
The lio_listio() function shall initiate a list of I/O requests with a single function call.

The mode argument takes one of the values LIO_WAIT or LIO_NOWAIT declared in <aio.h> and
determines whether the function returns when the I/O operations have been completed, or as
soon as the operations have been queued. If the mode argument is LIO_WAIT, the function shall
wait until all I/O is complete and the sig argument shall be ignored.

If the mode argument is LIO_NOWAIT, the function shall return immediately, and asynchronous
notification shall occur, according to the sig argument, when all the I/O operations complete. If
sig is NULL, then no asynchronous notification shall occur. If sig is not NULL, asynchronous
notification occurs as specified in Section 2.4.1 (on page 28) when all the requests in list have
completed.

The I/O requests enumerated by list are submitted in an unspecified order.

The list argument is an array of pointers to aiocb structures. The array contains nent elements.
The array may contain NULL elements, which shall be ignored.

The aio_lio_opcode field of each aiocb structure specifies the operation to be performed. The
supported operations are LIO_READ, LIO_WRITE, and LIO_NOP; these symbols are defined in
<aio.h>. The LIO_NOP operation causes the list entry to be ignored. If the aio_lio_opcode
element is equal to LIO_READ, then an I/O operation is submitted as if by a call to aio_read()
with the aiocbp equal to the address of the aiocb structure. If the aio_lio_opcode element is equal
to LIO_WRITE, then an I/O operation is submitted as if by a call to aio_write() with the aiocbp
equal to the address of the aiocb structure.

The aio_fildes member specifies the file descriptor on which the operation is to be performed.
The aio_buf member specifies the address of the buffer to or from which the data is transferred.
The aio_nbytes member specifies the number of bytes of data to be transferred.
The members of the aiocb structure further describe the I/O operation to be performed, in a
manner identical to that of the corresponding aiocb structure when used by the aio_read() and
aio_write() functions.

The nent argument specifies how many elements are members of the list; that is, the length of the
array.

The behavior of this function is altered according to the definitions of synchronized I/O data
integrity completion and synchronized I/O file integrity completion if synchronized I/O is
enabled on the file associated with aio_fildes.

For regular files, no data transfer shall occur past the offset maximum established in the open
file description associated with aiocbp->aio_fildes.
RETURN VALUE

If the mode argument has the value LIO_NOWAIT, the lio_listio() function shall return the value
zero if the I/O operations are successfully queued; otherwise, the function shall return the value
−1 and set errno to indicate the error.

If the mode argument has the value LIO_WAIT, the lio_listio() function shall return the value
zero when all the indicated I/O has completed successfully. Otherwise, lio_listio() shall return a
value of −1 and set errno to indicate the error.

In either case, the return value only indicates the success or failure of the lio_listio() call itself,
not the status of the individual I/O requests. In some cases one or more of the I/O requests
contained in the list may fail. Failure of an individual request does not prevent completion of
any other individual request. To determine the outcome of each I/O request, the application
shall examine the error status associated with each aiocb control block. The error statuses so
returned are identical to those returned as the result of an aio_read() or aio_write() function.

ERRORS

The lio_listio() function shall fail if:

[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.

[EAGAIN] The number of entries indicated by nent would cause the system-wide limit
[AIO_MAX] to be exceeded.

[EINVAL] The mode argument is not a proper value, or the value of nent was greater than
[AIO_LISTIO_MAX].

[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
lio_listio() 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.

[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.

In addition to the errors returned by the lio_listio() function, if the lio_listio() function succeeds
or fails with errors of [EAGAIN], [EINTR], or [EIO], then some of the I/O specified by the list
may have been initiated. If the lio_listio() function fails with an error code other than [EAGAIN],
[EINTR], or [EIO], no operations from the list shall have been initiated. The I/O operation
indicated by each list element can encounter errors specific to the individual read or write
function being performed. In this event, the error status for each aiocb control block contains the
associated error code. The error codes that can be set are the same as would be set by a read() or
write() function, with the following additional error codes possible:

[EAGAIN] The requested I/O operation was not queued due to resource limitations.

[ECANCELED] The requested I/O was canceled before the I/O completed due to an explicit
aio_cancel() request.

[EFBIG] The aiocbp->aio_lio_opcode is LIO_WRITE, the file is a regular file,
aiocbp->aio_nbytes is greater than 0, and the aiocbp->aio_offset is greater than or
equal to the offset maximum in the open file description associated with
 aiocbp->aio_fildes.

[EINPROGRESS] The requested I/O is in progress.

[EOVERFLOW] The aiocbp->aio_llo_opcode is LIO_READ, the file is a regular file, aiocbp->aio_nbbytes is greater than 0, and the aiocbp->aio_offset is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with aiocbp->aio_fildes.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
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_llo_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]).

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.

FUTURE DIRECTIONS
None.

SEE ALSO
aio_read(), aio_write(), aio_error(), aio_return(), aio_cancel(), close(), exec, exit(), fork(), lseek(), read(), the Base Definitions volume of IEEE Std 1003.1-2001, <aio.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.
Large File Summit extensions are added.

Issue 6
The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.
The lio_listio() function is marked as part of the Asynchronous Input and Output option.
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

• In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs past the offset maximum established in the open file description associated with aiocbp->aio_fildes. This change is to support large files.

• The [EBIG] and [EOVERFLOW] error conditions are defined. This change is to support large files.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The restrict keyword is added to the lio_listio() prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
listen — listen for socket connections and limit the queue of incoming connections

SYNOPSIS
#include <sys/socket.h>
int listen(int socket, int backlog);

DESCRIPTION
The listen() function shall mark a connection-mode socket, specified by the socket argument, as accepting connections.

The backlog argument provides a hint to the implementation which the implementation shall use to limit the number of outstanding connections in the socket’s listen queue. Implementations may impose a limit on backlog and silently reduce the specified value. Normally, a larger backlog argument value shall result in a larger or equal length of the listen queue. Implementations shall support values of backlog up to SOMAXCONN, defined in <sys/socket.h>.

The implementation may include incomplete connections in its listen queue. The limits on the number of incomplete connections and completed connections queued may be different.

The implementation may have an upper limit on the length of the listen queue—either global or per accepting socket. If backlog exceeds this limit, the length of the listen queue is set to the limit.

If listen() is called with a backlog argument value that is less than 0, the function behaves as if it had been called with a backlog argument value of 0.

A backlog argument of 0 may allow the socket to accept connections, in which case the length of the listen queue may be set to an implementation-defined minimum value.

The socket in use may require the process to have appropriate privileges to use the listen() function.

RETURN VALUE
Upon successful completions, listen() shall return 0; otherwise, −1 shall be returned and errno set to indicate the error.

ERRORS
The listen() function shall fail if:

[EBADF] The socket argument is not a valid file descriptor.
[EDESTADDRREQ] The socket is not bound to a local address, and the protocol does not support listening on an unbound socket.
[EINVAL] The socket is already connected.
[ENOTSOCK] The socket argument does not refer to a socket.
[EOPNOTSUPP] The socket protocol does not support listen().

The listen() function may fail if:

[EACCES] The calling process does not have the appropriate privileges.
[EINVAL] The socket has been shut down.
[ENOBUFS] Insufficient resources are available in the system to complete the call.
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
accept(), connect(), socket(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

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

The DESCRIPTION is updated to describe the relationship of SOMAXCONN and the backlog argument.
NAME
llabs — return a long integer absolute value

SYNOPSIS
#include <stdlib.h>
long long llabs(long long i);

DESCRIPTION
Refer to labs().
NAME
lldiv — compute quotient and remainder of a long division

SYNOPSIS
#include <stdlib.h>

lldiv_t lldiv(long long numer, long long denom);

DESCRIPTION
Refer to ldiv().
NAME
llrint, llrintf, llrintl, — round to the nearest integer value using current rounding direction

SYNOPSIS
#include <math.h>

long long llrint(double x);
long long llrintf(float x);
long long llrintl(long double x);

DESCRIPTION
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.

These functions shall round their argument to the nearest integer value, rounding according to the current rounding direction.

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.

RETURN VALUE
Upon successful completion, these functions shall return the rounded integer value.

If x is NaN, a domain error shall occur, and an unspecified value is returned.
If x is +Inf, a domain error shall occur and an unspecified value is returned.
If x is −Inf, a domain error shall occur and an unspecified value is returned.
If the correct value is positive and too large to represent as a long long, a domain error shall occur and an unspecified value is returned.
If the correct value is negative and too large to represent as a long long, a domain error shall occur and an unspecified value is returned.

ERRORS
These functions shall fail if:

Domain Error The x argument is NaN or ±Inf, or the correct value is not representable as an integer.
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.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
These functions provide floating-to-integer conversions. They round according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and the invalid floating-point exception is raised. When they raise no other floating-point exception and the result differs from the argument, they raise the inexact
floating-point exception.

**FUTURE DIRECTIONS**
None.

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

**CHANGE HISTORY**
NAME
llround, llroundf, llroundl, — round to nearest integer value

SYNOPSIS
#include <math.h>
long long llround(double x);
long long llroundf(float x);
long long llroundl(long double x);

DESCRIPTION
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.

These functions shall round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction.

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.

RETURN VALUE
Upon successful completion, these functions shall return the rounded integer value.

If x is NaN, a domain error shall occur, and an unspecified value is returned.
If x is +Inf, a domain error shall occur and an unspecified value is returned.
If x is −Inf, a domain error shall occur and an unspecified value is returned.

If the correct value is positive and too large to represent as a long long, a domain error shall occur and an unspecified value is returned.
If the correct value is negative and too large to represent as a long long, a domain error shall occur and an unspecified value is returned.

ERRORS
These functions shall fail if:

Domain Error The x argument is NaN or ±Inf, or the correct value is not representable as an integer.

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.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
These functions differ from the lrint() functions in that the default rounding direction for the llround() functions round halfway cases away from zero and need not raise the inexact floating-point exception for non-integer arguments that round to within the range of the return type.
FUTURE DIRECTIONS
None.

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

CHANGE HISTORY
localeconv() — return locale-specific information

#include <locale.h>

struct lconv *localeconv(void);

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 localeconv() function shall set the components of an object with the type struct lconv with the values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The members of the structure with type char * are pointers to strings, any of which (except decimal_point) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type char are non-negative numbers, any of which can be {CHAR_MAX} to indicate that the value is not available in the current locale.

The members include the following:

char *decimal_point
The radix character used to format non-monetary quantities.

char *thousands_sep
The character used to separate groups of digits before the decimal-point character in formatted non-monetary quantities.

char *grouping
A string whose elements taken as one-byte integer values indicate the size of each group of digits in formatted non-monetary quantities.

char *int_curr_symbol
The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in the ISO 4217:1995 standard. The fourth character (immediately preceding the null byte) is the character used to separate the international currency symbol from the monetary quantity.

char *currency_symbol
The local currency symbol applicable to the current locale.

char *mon_decimal_point
The radix character used to format monetary quantities.

char *mon_thousands_sep
The separator for groups of digits before the decimal-point in formatted monetary quantities.

char *mon_grouping
A string whose elements taken as one-byte integer values indicate the size of each group of digits in formatted monetary quantities.

char *positive_sign
The string used to indicate a non-negative valued formatted monetary quantity.
char *negative_sign
   The string used to indicate a negative valued formatted monetary quantity.

char int_frac_digits
   The number of fractional digits (those after the decimal-point) to be displayed in an
   internationally formatted monetary quantity.

char frac_digits
   The number of fractional digits (those after the decimal-point) to be displayed in a
   formatted monetary quantity.

char p_cs_precedes
   Set to 1 if the currency_symbol or int_curr_symbol precedes the value for a non-negative
   formatted monetary quantity. Set to 0 if the symbol succeeds the value.

char p_sep_by_space
   Set to 0 if no space separates the currency_symbol or int_curr_symbol from the value for a
   non-negative internationally formatted monetary quantity. Set to 1 if a space separates the symbol from the
   value; and set to 2 if a space separates the symbol and the sign string, if adjacent.

char n_cs_precedes
   Set to 1 if the currency_symbol or int_curr_symbol precedes the value for a negative
   formatted monetary quantity. Set to 0 if the symbol succeeds the value.

char n_sep_by_space
   Set to 0 if no space separates the currency_symbol or int_curr_symbol from the value for a
   negative internationally formatted monetary quantity. Set to 1 if a space separates the symbol from the
   value; and set to 2 if a space separates the symbol and the sign string, if adjacent.

char p_sign_posn
   Set to a value indicating the positioning of the positive_sign for a non-negative formatted
   monetary quantity.

char n_sign_posn
   Set to a value indicating the positioning of the negative_sign for a negative formatted
   monetary quantity.

char int_p_cs_precedes
   Set to 1 or 0 if the int_curr_symbol respectively precedes or succeeds the value for a non-
   negative internationally formatted monetary quantity.

char int_n_cs_precedes
   Set to 1 or 0 if the int_curr_symbol respectively precedes or succeeds the value for a
   negative internationally formatted monetary quantity.

char int_p_sep_by_space
   Set to a value indicating the separation of the int_curr_symbol, the sign string, and the
   value for a non-negative internationally formatted monetary quantity.

char int_n_sep_by_space
   Set to a value indicating the separation of the int_curr_symbol, the sign string, and the
   value for a negative internationally formatted monetary quantity.

char int_p_sign_posn
   Set to a value indicating the positioning of the positive_sign for a non-negative
   internationally formatted monetary quantity.

char int_n_sign_posn
   Set to a value indicating the positioning of the negative_sign for a negative internationally
localeconv()

formatted monetary quantity.

The elements of grouping and mon_grouping are interpreted according to the following:

{CHAR_MAX}  No further grouping is to be performed.

0          The previous element is to be repeatedly used for the remainder of the digits.

other      The integer value is the number of digits that comprise the current group. The
            next element is examined to determine the size of the next group of digits
            before the current group.

The values of p_sep_by_space, n_sep_by_space, int_p_sep_by_space, and int_n_sep_by_space
are interpreted according to the following:

0          No space separates the currency symbol and value.

1          If the currency symbol and sign string are adjacent, a space separates them from the value;
            otherwise, a space separates the currency symbol from the value.

2          If the currency symbol and sign string are adjacent, a space separates them; otherwise, a
            space separates the sign string from the value.

For int_p_sep_by_space and int_n_sep_by_space, the fourth character of int_curr_symbol is
used instead of a space.

The values of p_sign_posn, n_sign_posn, int_p_sign_posn, and int_n_sign_posn are
interpreted according to the following:

0          Parentheses surround the quantity and currency_symbol or int_curr_symbol.

1          The sign string precedes the quantity and currency_symbol or int_curr_symbol.

2          The sign string succeeds the quantity and currency_symbol or int_curr_symbol.

3          The sign string immediately precedes the currency_symbol or int_curr_symbol.

4          The sign string immediately succeeds the currency_symbol or int_curr_symbol.

The implementation shall behave as if no function in this volume of IEEE Std 1003.1-2001 calls
localeconv() .

CX       The localeconv() function need not be reentrant. A function that is not required to be reentrant is
          not required to be thread-safe.

RETURN VALUE

The localeconv() function shall return a pointer to the filled-in object. The application shall not
modify the structure pointed to by the return value which may be overwritten by a subsequent
call to localeconv(). In addition, calls to setlocale() with the categories LC_ALL, LC_MONETARY,
or LC_NUMERIC may overwrite the contents of the structure.

ERRORS

No errors are defined.
EXAMPLES

None.

APPLICATION USAGE

The following table illustrates the rules which may be used by four countries to format monetary quantities.

<table>
<thead>
<tr>
<th>Country</th>
<th>Positive Format</th>
<th>Negative Format</th>
<th>International Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Italy</td>
<td>L.1.230</td>
<td>–L.1.230</td>
<td>ITL.1.230</td>
</tr>
<tr>
<td>Netherlands</td>
<td>F 1.234,56</td>
<td>F ~ 1.234,56</td>
<td>NLG 1.234,56</td>
</tr>
<tr>
<td>Norway</td>
<td>kr1.234,56</td>
<td>kr1.234,56–</td>
<td>NOK 1.234,56</td>
</tr>
<tr>
<td>Switzerland</td>
<td>SFrs.1,234,56</td>
<td>SFrs.1,234,56C</td>
<td>CHF 1,234.56</td>
</tr>
</tbody>
</table>

For these four countries, the respective values for the monetary members of the structure returned by `localeconv()` are:

<table>
<thead>
<tr>
<th></th>
<th>Italy</th>
<th>Netherlands</th>
<th>Norway</th>
<th>Switzerland</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int_curr_symbol</code></td>
<td>&quot;ITL.&quot;</td>
<td>&quot;NLG&quot;</td>
<td>&quot;NOK&quot;</td>
<td>&quot;CHF&quot;</td>
</tr>
<tr>
<td><code>currency_symbol</code></td>
<td>&quot;L.&quot;</td>
<td>&quot;F&quot;</td>
<td>&quot;kr&quot;</td>
<td>&quot;SFrs.&quot;</td>
</tr>
<tr>
<td><code>mon_decimal_point</code></td>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td><code>mon_thousands_sep</code></td>
<td>&quot;\3&quot;</td>
<td>&quot;\3&quot;</td>
<td>&quot;\3&quot;</td>
<td>&quot;\3&quot;</td>
</tr>
<tr>
<td><code>mon_grouping</code></td>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td><code>positive_sign</code></td>
<td>&quot;_&quot;</td>
<td>&quot;_&quot;</td>
<td>&quot;_&quot;</td>
<td>&quot;C&quot;</td>
</tr>
<tr>
<td><code>negative_sign</code></td>
<td>&quot;_&quot;</td>
<td>&quot;_&quot;</td>
<td>&quot;_&quot;</td>
<td>&quot;_&quot;</td>
</tr>
<tr>
<td><code>int_frac_digits</code></td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><code>frac_digits</code></td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><code>p_cs_precedes</code></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>p_sep_by_space</code></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>n_cs_precedes</code></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>n_sep_by_space</code></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>p_sign_posn</code></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>n_sign_posn</code></td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><code>int_p_cs_precedes</code></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>int_n_cs_precedes</code></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>int_p_sep_by_space</code></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>int_n_sep_by_space</code></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>p_sign_posn</code></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>n_sign_posn</code></td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`isalpha()`, `isascii()`, `nl_langinfo()`, `printf()`, `scanf()`, `setlocale()`, `strcat()`, `strchr()`, `strcmp()`, `strcoll()`, `strncpy()`, `strpbrk()`, `strsplit()`, `strtok()`, `strxfrm()`, `strtod()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<langinfo.h>`, `<locale.h>`

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.
localeconv()

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

The RETURN VALUE section is rewritten to avoid use of the term “must”.

This reference page is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME
localtime, localtime_r — convert a time value to a broken-down local time

SYNOPSIS
#include <time.h>

struct tm *localtime(const time_t *timer);

struct tm *localtime_r(const time_t *restrict timer,
                     struct tm *restrict result);

DESCRIPTION
For localtime(): 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 localtime() function shall convert the time in seconds since the Epoch pointed to by timer into a broken-down time, expressed as a local time. The function corrects for the timezone and any seasonal time adjustments. Local timezone information is used as though localtime() calls tzset().

The relationship between a time in seconds since the Epoch used as an argument to localtime() 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) corrected for timezone and any seasonal time adjustments, where the names in the structure and in the expression correspond.

The same relationship shall apply for localtime_r().

The localtime() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

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.

The localtime_r() function shall convert the time in seconds since the Epoch pointed to by timer into a broken-down time stored in the structure to which result points. The localtime_r() function shall also return a pointer to that same structure.

Unlike localtime(), the reentrant version is not required to set tzname.

RETURN VALUE
The localtime() function shall return a pointer to the broken-down time structure.

Upon successful completion, localtime_r() shall return a pointer to the structure pointed to by the argument result.

ERRORS
No errors are defined.
EXAMPLES

Getting the Local Date and Time

The following example uses the `time()` function to calculate the time elapsed, in seconds, since January 1, 1970 0:00 UTC (the Epoch), `localtime()` to convert that value to a broken-down time, and `asctime()` to convert the broken-down time values into a printable string.

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

int main(void)
{
    time_t result;
    result = time(NULL);
    printf("%s%ju secs since the Epoch\n", 
        asctime(localtime(&result)),
        (uintmax_t)result);
    return(0);
}
```

This example writes the current time to `stdout` in a form like this:

```
835810335 secs since the Epoch
```

Getting the Modification Time for a File

The following example gets the modification time for a file. The `localtime()` function converts the `time_t` value of the last modification date, obtained by a previous call to `stat()`, into a `tm` structure that contains the year, month, day, and so on.

```c
#include <time.h>
...
struct stat statbuf;
...
tm = localtime(&statbuf.st_mtime);
...
```

Timing an Event

The following example gets the current time, converts it to a string using `localtime()` and `asctime()`, and prints it to standard output using `fputs()`. It then prints the number of minutes to an event being timed.

```c
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
fputs(asctime(localtime(&now)), stdout);
printf("There are still %d minutes to the event.\n", 
```
APPLICATION USAGE
The `localtime_r()` function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
`asctime()`, `clock()`, `ctime()`, `difftime()`, `getdate()`, `gmtime()`, `mktime()`, `strftime()`, `strptime()`, `time()`, `utime()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<time.h>`

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
A note indicating that the `localtime()` function need not be reentrant is added to the DESCRIPTION.
The `localtime_r()` function is included for alignment with the POSIX Threads Extension.

Issue 6
The `localtime_r()` function is marked as part of the Thread-Safe Functions option.
Extensions beyond the ISO C standard are marked.
The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.
The `restrict` keyword is added to the `localtime_r()` prototype for alignment with the ISO/IEC 9899:1999 standard.
Examples are added.
NAME
lockf — record locking on files

SYNOPSIS
XSI
#include <unistd.h>

int lockf(int fildes, int function, off_t size);

DESCRIPTION
The lockf() function shall lock sections of a file with advisory-mode locks. Calls to lockf() from
tother 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_ULOCK</td>
<td>Unlock locked sections.</td>
</tr>
<tr>
<td>F_LOCK</td>
<td>Lock a section for exclusive use.</td>
</tr>
<tr>
<td>F_TLOCK</td>
<td>Test and lock a section for exclusive use.</td>
</tr>
<tr>
<td>F_TEST</td>
<td>Test a section for locks by other processes.</td>
</tr>
</tbody>
</table>

F_TEST shall detect if a lock by another process is present on the specified section.
F_LOCK and F_TLOCK shall both lock a section of a file if the section is available.
F_ULOCK shall remove locks from a section of the file.

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.

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.

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.

File locks shall be released on first close by the locking process of any file descriptor for the file.
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
section shall cause the remaining locked beginning and end portions to become two separate
locked sections. If the request would cause the number of locks in the system to exceed a
system-imposed limit, the request shall fail.

A potential for deadlock occurs if the threads of a process controlling a locked section are
blocked by accessing another process' locked section. If the system detects that deadlock would
occur, lockf() shall fail with an [EDEADLK] error.

The interaction between fcntl() and lockf() locks is unspecified.

Blocking on a section shall be interrupted by any signal.

An F_ULOCK request in which size is non-zero and the offset of the last byte of the requested
section is the maximum value for an object of type off_t, when the process has an existing lock
in which size is 0 and which includes the last byte of the requested section, shall be treated as a
request to unlock from the start of the requested section with a size equal to 0. Otherwise, an
F_ULOCK request shall attempt to unlock only the requested section.

Attempting to lock a section of a file that is associated with a buffered stream produces
unspecified results.

RETURN VALUE

Upon successful completion, lockf() shall return 0. Otherwise, it shall return −1, set errno to
indicate an error, and existing locks shall not be changed.

ERRORS

The lockf() function shall fail if:

[EBADF] The fildes argument is not a valid open file descriptor; or function is F_LOCK
or F_TLOCK and fildes is not a valid file descriptor open for writing.

[EACCES] or [EAGAIN]
The function argument is F_TLOCK or F_TEST and the section is already
locked by another process.

[EDEADLK] The function argument is F_LOCK and a deadlock is detected.

[EINVAL]
The function argument is not one of F_LOCK, F_TLOCK, F_TEST, or
F_ULOCK; or size plus the current file offset is less than 0.

[EOVERFLOW] The offset of the first, or if size is not 0 then the last, byte in the requested
section cannot be represented correctly in an object of type off_t.

The lockf() function may fail if:

[EAGAIN] The function argument is F_LOCK or F_TLOCK and the file is mapped with
mmap().

[EDEADLK] or [ENOLCK]
The function argument is F_LOCK, F_TLOCK, or F_ULOCK, and the request
would cause the number of locks to exceed a system-imposed limit.

[EOPNOTSUPP] or [EINVAL]
The implementation does not support the locking of files of the type indicated
by the fildes argument.
EXAMPLES

Locking a Portion of a File

In the following example, a file named `/home/cnd/mod1` is being modified. Other processes that use locking are prevented from changing it during this process. Only the first 10,000 bytes are locked, and the lock call fails if another process has any part of this area locked already.

```
#include <fcntl.h>
#include <unistd.h>

int fildes;
int status;
...

fildes = open("/home/cnd/mod1", O_RDWR);
status = lockf(fildes, F_TLOCK, (off_t)10000);
```

APPLICATION USAGE

Record-locking should not be used in combination with the `fopen()`, `fread()`, `fwrite()`, and other `stdio` functions. Instead, the more primitive, non-buffered functions (such as `open()`) should be used. Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The `stdio` functions are the most common source of unexpected buffering.

The `alarm()` function may be used to provide a timeout facility in applications requiring it.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`alarm()`, `chmod()`, `close()`, `creat()`, `fcntl()`, `fopen()`, `mmap()`, `open()`, `read()`, `write()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<unistd.h>`

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Large File Summit extensions are added. In particular, the description of `[EINVAL]` is clarified and moved from optional to mandatory status.

A note is added to the DESCRIPTION indicating the effects of attempting to lock a section of a file that is associated with a buffered stream.

Issue 6

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
   log, logf, logl — natural logarithm function

SYNOPSIS
   #include <math.h>
   double log(double x);
   float logf(float x);
   long double logl(long double x);

DESCRIPTION
   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

   These functions shall compute the natural logarithm of their argument x, \log_e(x).

   An application wishing to check for error situations should set errno to zero and call
   fe Cearexcet(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.

RETURN VALUE
   Upon successful completion, these functions shall return the natural logarithm of x.

   If x is ±0, a pole error shall occur and log(), logf(), and logl() shall return −HUGE_VAL,
   −HUGE_VALF, and −HUGE_VALL, respectively.

   For finite values of x that are less than 0, or if x is −Inf, a domain error shall occur, and either a
   NaN (if supported), or an implementation-defined value shall be returned.

   If x is NaN, a NaN shall be returned.

   If x is 1, +0 shall be returned.

   If x is +Inf, x shall be returned.

ERRORS
   These functions shall fail if:

   Domain Error   The finite value of x is negative, or x is −Inf.

   Pole Error     The value of x is zero.

   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.

   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.
EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
exp(), feclearexcept(), fetestexcept(), isnan(), log10(), log1p(), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The logf() and logl() functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.
NAME
log10, log10f, log10l — base 10 logarithm function

SYNOPSIS
#include <math.h>

double log10(double x);
float log10f(float x);
long double log10l(long double x);

DESCRIPTION
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.

These functions shall compute the base 10 logarithm of their argument \( x \), \( \log_{10}(x) \).

An application wishing to check for error situations should set \( \text{errno} \) to zero and call \( \text{fclearexcept}(\text{FE_ALL_EXCEPT}) \) before calling these functions. On return, if \( \text{errno} \) is non-zero or \( \text{fetestexcept}(\text{FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW}) \) is non-zero, an error has occurred.

RETURN VALUE
Upon successful completion, these functions shall return the base 10 logarithm of \( x \).

If \( x \) is \( \pm 0 \), a pole error shall occur and \( \log10() \), \( \log10f() \), and \( \log10l() \) shall return \(-\text{HUGE_VAL} \), \(-\text{HUGE_VALF} \), and \(-\text{HUGE_VALL} \), respectively.

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.

If \( x \) is \( \text{NaN} \), a NaN shall be returned.

If \( x \) is 1, +0 shall be returned.

If \( x \) is +\( \text{Inf} \), +\( \text{Inf} \) shall be returned.

ERRORS
These functions shall fail if:

Domain Error  The finite value of \( x \) is negative, or \( x \) is \(-\text{Inf} \).

If the integer expression (\( \text{math_errhandling} \) \& \text{MATH_ERRNO}) is non-zero, then \( \text{errno} \) 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.

Pole Error  The value of \( x \) is zero.

If the integer expression (\( \text{math_errhandling} \) \& \text{MATH_ERRNO}) is non-zero, then \( \text{errno} \) shall be set to [ERANGE]. If the integer expression (\( \text{math_errhandling} \) \& \text{MATH_ERREXCEPT}) is non-zero, then the divide-by-zero floating-point exception shall be raised.
EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
feclearexcept(), fetestexcept(), isnan(), log(), pow(), the Base Definitions volume of
IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
<math.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This
text was previously published in the APPLICATION USAGE section.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The log10f() and log10l() functions are added for alignment with the ISO/IEC 9899:1999
standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
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.
NAME
  log1p, log1pf, log1pl — compute a natural logarithm

SYNOPSIS
  #include <math.h>
  double log1p(double x);
  float log1pf(float x);
  long double log1pl(long double x);

DESCRIPTION
  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

  These functions shall compute \( \log_e(1.0 + x) \).

  An application wishing to check for error situations should set \( \text{errno} \) to zero and call
  \( \text{fencearexcept}(\text{FE_ALL_EXCEPT}) \) before calling these functions. On return, if \( \text{errno} \) is
  non-zero or \( \text{fetestexcept}(\text{FE_INVALID} | \text{FE_DIVBYZERO} | \text{FE_OVERFLOW} | \text{FE_UNDERFLOW}) \) is non-
  zero, an error has occurred.

RETURN VALUE
  Upon successful completion, these functions shall return the natural logarithm of \( 1.0 + x \).

  If \( x \) is \(-1\), a pole error shall occur and \( \log1p() \), \( \log1pf() \), and \( \log1pl() \) shall return \(-\text{HUGE}_\text{VAL} \),
  \(-\text{HUGE}_\text{VALF} \), and \(-\text{HUGE}_\text{VALL} \), respectively.

  For finite values of \( x \) that are less than \(-1\), or if \( x \) is \(-\text{Inf} \), a domain error shall occur, and either a
  \( \text{NaN} \) (if supported), or an implementation-defined value shall be returned.

  If \( x \) is \( \text{NaN} \), a \( \text{NaN} \) shall be returned.

  If \( x \) is \( \pm0 \), or \( +\text{Inf} \), \( x \) shall be returned.

  If \( x \) is subnormal, a range error may occur and \( x \) should be returned.

ERRORS
  These functions shall fail if:

  Domain Error  The finite value of \( x \) is less than \(-1 \), or \( x \) is \(-\text{Inf} \).

  If the integer expression (\( \text{math_errhandling} \& \text{MATH_ERRNO} \)) is non-zero,
  then \( \text{errno} \) shall be set to [\( \text{EDOM} \)]. If the integer expression (\( \text{math_errhandling} \& \text{MATH_ERRNO} \& \text{MATH_ERREXCEPT} \)) is non-zero, then the invalid floating-point exception
  shall be raised.

  Pole Error  The value of \( x \) is \(-1 \).

  If the integer expression (\( \text{math_errhandling} \& \text{MATH_ERRNO} \)) is non-zero,
  then \( \text{errno} \) shall be set to [\( \text{ERANGE} \)]. If the integer expression
  (\( \text{math_errhandling} \& \text{MATH_ERRNO} \& \text{MATH_ERREXCEPT} \)) is non-zero, then the divide-by-zero floating-point exception shall be raised.

  These functions may fail if:

  Range Error  The value of \( x \) is subnormal.

  If the integer expression (\( \text{math_errhandling} \& \text{MATH_ERRNO} \)) is non-zero,
  then \( \text{errno} \) shall be set to [\( \text{ERANGE} \)]. If the integer expression
  (\( \text{math_errhandling} \& \text{MATH_ERRNO} \& \text{MATH_ERREXCEPT} \)) is non-zero, then the underflow
  floating-point exception shall be raised.
EXAMPLES
None.

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

RATIONALE
None.

FUTURE DIRECTIONS
None.

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

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
The DESCRIPTION is updated to avoid use of the term "must" for application requirements.
The log1p() function is no longer marked as an extension.
The log1pf() and log1pl() functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.
NAME
log2, log2f, log2l — compute base 2 logarithm functions

SYNOPSIS
#include <math.h>

double log2(double x);
float log2f(float x);
long double log2l(long double x);

DESCRIPTION
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.

These functions shall compute the base 2 logarithm of their argument \( x \), \( \log_2(x) \).

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

RETURN VALUE
Upon successful completion, these functions shall return the base 2 logarithm of \( x \).

If \( x \) is \( \pm 0 \), a pole error shall occur and \( \log2() \), \( \log2f() \), and \( \log2l() \) shall return \( -\text{HUGE VAL} \), \( -\text{HUGE VALF} \), and \( -\text{HUGE VALL} \), respectively.

For finite values of \( x \) that are less than 0, or if \( x \) is \( -\text{Inf} \), a domain error shall occur, and either a \( \text{NaN} \) (if supported), or an implementation-defined value shall be returned.

If \( x \) is \( \text{NaN} \), a \( \text{NaN} \) shall be returned.

If \( x \) is 1, +0 shall be returned.

If \( x \) is \( +\text{Inf} \), \( x \) shall be returned.

ERRORS
These functions shall fail if:

Domain Error The finite value of \( x \) is less than zero, or \( x \) is \( -\text{Inf} \).

If the integer expression (\( \text{math_errhandling} & \text{MATH_ERRNO} \)) is non-zero, then \( \text{errno} \) shall be set to \[ \text{EDOM} \]. If the integer expression (\( \text{math_errhandling} & \text{MATH_ERREXCEPT} \)) is non-zero, then the invalid floating-point exception shall be raised.

Pole Error The value of \( x \) is zero.

If the integer expression (\( \text{math_errhandling} & \text{MATH_ERRNO} \)) is non-zero, then \( \text{errno} \) shall be set to \[ \text{ERANGE} \]. If the integer expression (\( \text{math_errhandling} & \text{MATH_ERREXCEPT} \)) is non-zero, then the divide-by-zero floating-point exception shall be raised.
EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

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

CHANGE HISTORY
NAME
logb, logbf, logbl — radix-independent exponent

SYNOPSIS
#include <math.h>

double logb(double x);
float logbf(float x);
long double logbl(long double x);

DESCRIPTION
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.

These functions shall compute the exponent of \( x \), which is the integral part of \( \log_r |x| \), as a signed floating-point value, for non-zero \( x \), where \( r \) is the radix of the machine’s floating-point arithmetic, which is the value of FLT_RADIX defined in the <float.h> header.

If \( x \) is subnormal it is treated as though it were normalized; thus for finite positive \( x \):

\[
1 \leq x \cdot FLT\_RADIX^{-\text{logb}(x)} < \text{FLT\_RADIX}
\]

An application wishing to check for error situations should set \textit{errno} to zero and call \textit{f clearexcept}(FE\_ALL\_EXCEPT) before calling these functions. On return, if \textit{errno} is non-zero or \textit{fetestexcept}(FE\_INVALID | FE\_DIVBYZERO | FE\_OVERFLOW | FE\_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE
Upon successful completion, these functions shall return the exponent of \( x \).

If \( x \) is \( \pm 0 \), a pole error shall occur and \textit{logb()}, \textit{logbf()}, and \textit{logbl()} shall return \( -\text{HUGE\_VAL} \), \( -\text{HUGE\_VALF} \), and \( -\text{HUGE\_VALL} \), respectively.

If \( x \) is NaN, a NaN shall be returned.

If \( x \) is \( \pm \text{Inf} \), +Inf shall be returned.

ERRORS
These functions shall fail if:

Pole Error The value of \( x \) is \( \pm 0 \).

If the integer expression (math\_errhandling & MATH\_ERRNO) is non-zero, then \textit{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.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.
FUTURE DIRECTIONS
None.

SEE ALSO
fesetexcept(), 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>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
The logb() function is no longer marked as an extension.
The logbf() and logbl() functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.
NAME
logf, logl — natural logarithm function

SYNOPSIS
#include <math.h>

float logf(float x);
long double logl(long double x);

DESCRIPTION
Refer to log().
NAME
longjmp — non-local goto

SYNOPSIS
#include <setjmp.h>

void longjmp(jmp_buf env, int val);

DESCRIPTION
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

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.

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.

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.

The effect of a call to longjmp() where initialization of the jmp_buf structure was not performed
in the calling thread is undefined.

RETURN VALUE
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.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
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).
**longjmp()**

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`setjmp()`, `sigaction()`, `siglongjmp()`, `sigsetjmp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<setjmp.h>`

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**
The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

**Issue 6**
Extensions beyond the ISO C standard are marked.

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

• The DESCRIPTION now explicitly makes `longjmp()`'s effect on the signal mask unspecified.

The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
lrand48 — generate uniformly distributed pseudo-random non-negative long integers

SYNOPSIS
XSI
#include <stdlib.h>

long lrand48(void);

DESCRIPTION
Refer to drand48().
NAME
lrint, lrintf, lrintl — round to nearest integer value using current rounding direction

SYNOPSIS
#include <math.h>

long lrint(double x);
long lrintf(float x);
long lrintl(long double x);

DESCRIPTION
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

These functions shall round their argument to the nearest integer value, rounding according to
the current rounding direction.

An application wishing to check for error situations should set errno to zero and call
fclear except(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.

RETURN VALUE
Upon successful completion, these functions shall return the rounded integer value.

If x is NaN, a domain error shall occur and an unspecified value is returned.
If x is +Inf, a domain error shall occur and an unspecified value is returned.
If x is −Inf, a domain error shall occur and an unspecified value is returned.
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.
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.

ERRORS
These functions shall fail if:

Domain Error The x argument is NaN or ±Inf, or the correct value is not representable as an
integer.

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.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
These functions provide floating-to-integer conversions. They round according to the current
rounding direction. If the rounded value is outside the range of the return type, the numeric
result is unspecified and the invalid floating-point exception is raised. When they raise no other
floating-point exception and the result differs from the argument, they raise the inexact
floating-point exception.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

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

**CHANGE HISTORY**

NAME
lround, lroundf, lroundl — round to nearest integer value

SYNOPSIS
#include <math.h>

long lround(double x);
long lroundf(float x);
long lroundl(long double x);

DESCRIPTION
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.

These functions shall round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction.

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.

RETURN VALUE
Upon successful completion, these functions shall return the rounded integer value.

If x is NaN, a domain error shall occur and an unspecified value is returned.
If x is ±Inf, a domain error shall occur and an unspecified value is returned.
If x is −Inf, a domain error shall occur and an unspecified value is returned.
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.
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.

ERRORS
These functions shall fail if:

Domain Error The x argument is NaN or ±Inf, or the correct value is not representable as an integer.

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.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
These functions differ from the lrint() functions in the default rounding direction, with the lround() functions rounding halfway cases away from zero and needing not to raise the inexact floating-point exception for non-integer arguments that round to within the range of the return type.
FUTURE DIRECTIONS

None.

SEE ALSO

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

CHANGE HISTORY

NAME
lsearch, lfind — linear search and update

SYNOPSIS

```c
#include <search.h>

void *lsearch(const void *key, void *base, size_t *nelp, size_t width,
               int (*compar)(const void *, const void *));
void *lfind(const void *key, const void *base, size_t *nelp,
            size_t width, int (*compar)(const void *, const void *));
```

DESCRIPTION
The `lsearch()` function shall linearly search the table and return a pointer into the table for the
matching entry. If the entry does not occur, it shall be added at the end of the table. The `key`
argument points to the entry to be sought in the table. The `base` argument points to the first
element in the table. The `width` argument is the size of an element in bytes. The `nelp` argument
points to an integer containing the current number of elements in the table. The integer to which
`nelp` points shall be incremented if the entry is added to the table. The `compar` argument points to
a comparison function which the application shall supply (for example, `strcmp()`). It is called
with two arguments that point to the elements being compared. The application shall ensure
that the function returns 0 if the elements are equal, and non-zero otherwise.

The `lfind()` function shall be equivalent to `lsearch()`, except that if the entry is not found, it is not
added to the table. Instead, a null pointer is returned.

RETURN VALUE
If the searched for entry is found, both `lsearch()` and `lfind()` shall return a pointer to it. Otherwise,
`lfind()` shall return a null pointer and `lsearch()` shall return a pointer to the newly added element.

Both functions shall return a null pointer in case of error.

ERRORS
No errors are defined.

EXAMPLES
Storing Strings in a Table
This fragment reads in less than or equal to TABSIZE strings of length less than or equal to
ELSIZE and stores them in a table, eliminating duplicates.

```c
#include <stdio.h>
#include <string.h>
#include <search.h>
#define TABSIZE 50
#define ELSIZE 120
...
char line[ELSIZE], tab[TABSIZE][ELSIZE];
size_t nel = 0;
...
while (fgets(line, ELSIZE, stdin) != NULL && nel < TABSIZE)
  (void) lsearch(line, tab, &nel,
                 ELSIZE, (int (*)(const void *, const void *)) strcmp);
...
Finding a Matching Entry
The following example finds any line that reads "This is a test.".

```
#include <search.h>
#include <string.h>
...
char line[ELSIZE], tab[TABSIZE][ELSIZE];
size_t nel = 0;
char *findline;
void *entry;
findline = "This is a test.\n";
entry = lfind(findline, tab, &nel, ELSIZE, (int (*)(const void *, const void *)) strcmp);
```

APPLICATION USAGE
The comparison function need not compare every byte, so arbitrary data may be contained in
the elements in addition to the values being compared.
Undefined results can occur if there is not enough room in the table to add a new item.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
hcreate(), tsearch(), the Base Definitions volume of IEEE Std 1003.1-2001, <search.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The DESCRIPTION is updated to avoid use of the term "must" for application requirements.
NAME
lseek — move the read/write file offset

SYNOPSIS
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);

DESCRIPTION
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.
The symbolic constants SEEK_SET, SEEK_CUR, and SEEK_END are defined in <unistd.h>.
The behavior of lseek() on devices which are incapable of seeking is implementation-defined.
The value of the file offset associated with such a device is undefined.
The lseek() function shall allow the file offset to be set beyond the end of the existing data in the
file. If data is later written at this point, subsequent reads of data in the gap shall return bytes
with the value 0 until data is actually written into the gap.
The lseek() function shall not, by itself, extend the size of a file.

SHM
If fildes refers to a shared memory object, the result of the lseek() function is unspecified.

TYM
If fildes refers to a typed memory object, the result of the lseek() function is unspecified.

RETURN VALUE
Upon successful completion, the resulting offset, as measured in bytes from the beginning of the
file, shall be returned. Otherwise, (off_t)-1 shall be returned, errno shall be set to indicate the
error, and the file offset shall remain unchanged.

ERRORS
The lseek() function shall fail if:

[EBADF] The fildes argument is not an open file descriptor.
[EINVAL] The whence argument is not a proper value, or the resulting file offset would
be negative for a regular file, block special file, or directory.
[EOVERFLOW] The resulting file offset would be a value which cannot be represented
correctly in an object of type off_t.

[ESPIPE] The fildes argument is associated with a pipe, FIFO, or socket.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
The ISO C standard includes the functions fgetpos() and fsetpos(), which work on very large files
by use of a special positioning type.
Although lseek() may position the file offset beyond the end of the file, this function does not
itself extend the size of the file. While the only function in IEEE Std 1003.1-2001 that may directly
extend the size of the file is `write()`, `truncate()`, and `ftruncate()`, several functions originally derived from the ISO C standard, such as `fwrite()`, `fprintf()`, and so on, may do so (by causing calls on `write()`).

An invalid file offset that would cause `[EINVAL]` to be returned may be both implementation-defined and device-dependent (for example, memory may have few invalid values). A negative file offset may be valid for some devices in some implementations.

The POSIX.1-1990 standard did not specifically prohibit `lseek()` from returning a negative offset. Therefore, an application was required to clear `errno` prior to the call and check `errno` upon return to determine whether a return value of `(off_t)−1` is a negative offset or an indication of an error condition. The standard developers did not wish to require this action on the part of a conforming application, and chose to require that `errno` be set to `[EINVAL]` when the resulting file offset would be negative for a regular file, block special file, or directory.

### FUTURE DIRECTIONS
None.

### SEE ALSO
`open()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/types.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.
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.

- The `[EOVERFLOW]` error condition is added. This change is to support large files.

An additional `[ESPIPE]` error condition is added for sockets.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that `lseek()` results are unspecified for typed memory objects.
NAME
lstat — get symbolic link status

SYNOPSIS
#include <sys/stat.h>

int lstat(const char *restrict path, struct stat *restrict buf);

DESCRIPTION
The lstat() function shall be equivalent to stat(), except when path refers to a symbolic link. In that case lstat() shall return information about the link, while stat() shall return information about the file the link references.

For symbolic links, the st_mode member shall contain meaningful information when used with the file type macros, and the st_size member shall contain the length of the pathname contained in the symbolic link. File mode bits and the contents of the remaining members of the stat structure are unspecified. The value returned in the st_size member is the length of the contents of the symbolic link, and does not count any trailing null.

RETURN VALUE
Upon successful completion, lstat() shall return 0. Otherwise, it shall return −1 and set errno to indicate the error.

ERRORS
The lstat() function shall fail if:

[EACCES] A component of the path prefix denies search permission.

[EIO] An error occurred while reading from the file system.

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

[ENAMETOOLONG] The length of a pathname exceeds [PATH_MAX] or a pathname component is longer than [NAME_MAX].

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

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

[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.

The lstat() function may fail if:

[ELOOP] More than [SYMLOOP_MAX] symbolic links were encountered during resolution of the path argument.

[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].

[EOVERFLOW] One of the members is too large to store into the structure pointed to by the buf argument.
EXAMPLES

Obtaining Symbolic Link Status Information

The following example shows how to obtain status information for a symbolic link named /modules/pass1. The structure variable buffer is defined for the stat structure. If the path argument specified the filename for the file pointed to by the symbolic link (/home/cnd/mod1), the results of calling the function would be the same as those returned by a call to the stat() function.

```c
#include <sys/stat.h>

struct stat buffer;
int status;
...
status = lstat("/modules/pass1", &buffer);
```

APPLICATION USAGE

None.

RATIONALE

The lstat() function is not required to update the time-related fields if the named file is not a symbolic link. While the st_uid, st_gid, st_atime, st_mtime, and st_ctime members of the stat structure may apply to a symbolic link, they are not required to do so. No functions in IEEE Std 1003.1-2001 are required to maintain any of these time fields.

FUTURE DIRECTIONS

None.

SEE ALSO

fstat(), readlink(), stat(), symlink(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/stat.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Large File Summit extensions are added.

Issue 6

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

- This function is now mandatory.
- The [ELOOP] optional error condition is added.

The restrict keyword is added to the lstat() prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
makecontext, swapcontext — manipulate user contexts

SYNOPSIS
#include <ucontext.h>

void makecontext(ucontext_t *ucp, void (*func)(void),
    int argc, ...);
int swapcontext(ucontext_t *restrict oucp,
    const ucontext_t *restrict ucp);

DESCRIPTION
The makecontext() function shall modify the context specified by ucp, which has been initialized
using getcontext(). When this context is resumed using swapcontext() or setcontext(), program
execution shall continue by calling func, passing it the arguments that follow argc in the
makecontext() call.

Before a call is made to makecontext(), the application shall ensure that the context being
modified has a stack allocated for it. The application shall ensure that the value of argc matches
the number of integer arguments passed to func; otherwise, the behavior is undefined.

The uc_link member is used to determine the context that shall be resumed when the context
being modified by makecontext() returns. The application shall ensure that the uc_link member is
initialized prior to the call to makecontext().

The swapcontext() function shall save the current context in the context structure pointed to by
oucp and shall set the context to the context structure pointed to by ucp.

RETURN VALUE
Upon successful completion, swapcontext() shall return 0. Otherwise, −1 shall be returned and
errno set to indicate the error.

ERRORS
The swapcontext() function shall fail if:

[ENOMEM] The ucp argument does not have enough stack left to complete the operation.

EXAMPLES
The following example illustrates the use of makecontext():

#include <stdio.h>
#include <ucontext.h>
static ucontext_t ctx[3];
static void
f1 (void)
{
    puts("start f1");
    swapcontext(&ctx[1], &ctx[2]);
    puts("finish f1");
}

static void
f2 (void)
{
    puts("start f2");
    swapcontext(&ctx[2], &ctx[1]);
makecontext()

```c
int main (void)
{
    char st1[8192];
    char st2[8192];
    getcontext(&ctx[1]);
    ctx[1].uc_stack.ss_sp = st1;
    ctx[1].uc_stack.ss_size = sizeof st1;
    ctx[1].uc_link = &ctx[0];
    makecontext(&ctx[1], f1, 0);
    getcontext(&ctx[2]);
    ctx[2].uc_stack.ss_sp = st2;
    ctx[2].uc_stack.ss_size = sizeof st2;
    ctx[2].uc_link = &ctx[1];
    makecontext(&ctx[2], f2, 0);
    swapcontext(&ctx[0], &ctx[2]);
    return 0;
}
```

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
exit(), getcontext(), sigaction(), sigprocmask(), the Base Definitions volume of IEEE Std 1003.1-2001, <ucontext.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

In the ERRORS section, the description of [ENOMEM] is changed to apply to swapcontext() only.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The restrict keyword is added to the swapcontext() prototype for alignment with the ISO/IEC 9899:1999 standard.
**NAME**
malloc — a memory allocator

**SYNOPSIS**
```
#include <stdlib.h>

void *malloc(size_t size);
```

**DESCRIPTION**
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 `malloc()` function shall allocate unused space for an object whose size in bytes is specified by `size` and whose value is unspecified.

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.

**RETURN VALUE**
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.

**ERRORS**
The `malloc()` function shall fail if:

- `[ENOMEM]` Insufficient storage space is available.

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`calloc()`, `free()`, `realloc()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 6**
Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:
• In the RETURN VALUE section, the requirement to set `errno` to indicate an error is added.

• The [ENOMEM] error condition is added.
NAME
mblen — get number of bytes in a character

SYNOPSIS
#include <stdlib.h>
int mblen(const char *s, size_t n);

DESCRIPTION
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

If s is not a null pointer, mblen() shall determine the number of bytes constituting the character
pointed to by s. Except that the shift state of mbtowc() is not affected, it shall be equivalent to:

mbtowc((wchar_t *)0, s, n);

The implementation shall behave as if no function defined in this volume of
IEEE Std 1003.1-2001 calls mblen().

The behavior of this function is affected by the LC_CTYPE category of the current locale. For a
state-dependent encoding, this function shall be 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.

RETURN VALUE
If s is a null pointer, mblen() 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, mblen() shall
either return 0 (if s points to the null byte), or return the number of bytes that constitute the
character (if the next n or fewer bytes form a valid character), or return −1 (if they do not form a
valid character) and may set errno to indicate the error. In no case shall the value returned be
greater than n or the value of the [MB_CUR_MAX] macro.

ERRORS
The mblen() function may fail if:

[EILSEQ] Invalid character sequence is detected.

EXERCISES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO

mbtowc(), mbstowcs(), wcstomb(), wcstombs(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY

First released in Issue 4. Aligned with the ISO C standard.
NAME
mbrlen — get number of bytes in a character (restartable)

SYNOPSIS
#include <wchar.h>

size_t mbrlen(const char *restrict s, size_t n,
               mbstate_t *restrict ps);

DESCRIPTION
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.

If s is not a null pointer, mbrlen() shall determine the number of bytes constituting the character pointed to by s. It shall be equivalent to:

    mbstate_t internal;
    mbtowc(NULL, s, n, ps != NULL ? ps : &internal);

If ps is a null pointer, the mbrlen() 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 mbrlen().

The behavior of this function is affected by the LC_CTYPE category of the current locale.

RETURN VALUE
The mbrlen() function shall return the first of the following that applies:

    0 If the next n or fewer bytes complete the character that corresponds to the null wide character.

    positive If the next n or fewer bytes complete a valid character; the value returned shall be the number of bytes that complete the character.

    (size_t)−2 If the next n bytes contribute to an incomplete but potentially valid character, and all n bytes have been processed. When n has at least the value of the \[MB_CUR_MAX\] macro, this case can only occur if s points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

    (size_t)−1 If an encoding error occurs, in which case the next n or fewer bytes do not contribute to a complete and valid character. In this case, [EILSEQ] shall be stored in errno and the conversion state is undefined.

ERRORS
The mbrlen() function may fail if:

    [EINVAL] ps points to an object that contains an invalid conversion state.

    [EILSEQ] Invalid character sequence is detected.
mbrlen()

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
mbsinit(), mbtowc(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY

Issue 6
The mbrlen() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
mbrtowc( )

NAME
mbrtowc — convert a character to a wide-character code (restartable)

SYNOPSIS
#include <wchar.h>

size_t mbrtowc(wchar_t *restrict pwc, const char *restrict s,
                size_t n, mbstate_t *restrict ps);

DESCRIPTION
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.

If s is a null pointer, the mbrtowc( ) function shall be equivalent to the call:

mbrtowc(NULL, "", 1, ps)

In this case, the values of the arguments pwc and n are ignored.

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.

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().

The behavior of this function is affected by the LC_CTYPE category of the current locale.

RETURN VALUE
The mbrtowc() function shall return the first of the following that applies:

0 If the next n or fewer bytes complete the character that corresponds to the null wide character (which is the value stored).

between 1 and n inclusive

If the next n 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.

(size_t)-2 If the next n bytes contribute to an incomplete but potentially valid character, and all n bytes have been processed (no value is stored). When n has at least the value of the {MB_CUR_MAX} macro, this case can only occur if s points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

(size_t)-1 If an encoding error occurs, in which case the next n or fewer bytes do not contribute to a complete and valid character (no value is stored). In this case, [EILSEQ] shall be stored in errno and the conversion state is undefined.
Errors
The `mbrtowc()` function may fail if:

- `[EINVAL]` `ps` points to an object that contains an invalid conversion state.
- `[EILSEQ]` Invalid character sequence is detected.

Examples
None.

Application Usage
None.

Rationale
None.

Future Directions
None.

See Also
`mbsinit()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`

Change History
- Issue 6
  - The `mbrtowc()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
  - The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:
    - The `[EINVAL]` error condition is added.
NAME
mbsinit — determine conversion object status

SYNOPSIS
#include <wchar.h>
int mbsinit(const mbstate_t *ps);

DESCRIPTION
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.

If ps is not a null pointer, the mbsinit() function shall determine whether the object pointed to by ps describes an initial conversion state.

RETURN VALUE
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.

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.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
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.

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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
mbrlen(), mbtowc(), wctomb(), mbsrtowcs(), wcscrtombs(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
NAME
mbsrtowcs — convert a character string to a wide-character string (restartable)

SYNOPSIS
#include <wchar.h>

size_t mbsrtowcs(wchar_t *restrict dst, const char **restrict src,
                 size_t len, mbstate_t *restrict ps);

DESCRIPTION
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

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).

Each conversion shall take place as if by a call to the mbtowc() function.

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.

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().

The behavior of this function shall be affected by the LC_CTYPE category of the current locale.

RETURN VALUE
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).

ERRORS
The mbsrtowcs() function may fail if:

CX [EINVAL] ps points to an object that contains an invalid conversion state.

[CX] [EILSEQ] Invalid character sequence is detected.
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
mbsinit(), mbrtowc(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY

Issue 6
The mbstowcs() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
The [EINVAL] error condition is marked CX.
**NAME**

mbstowcs — convert a character string to a wide-character string

**SYNOPSIS**

```c
#include <stdlib.h>

size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s, size_t n);
```

**DESCRIPTION**

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 `mbstowcs()` function shall convert a sequence of characters that begins in the initial shift state from the array pointed to by `s` into a sequence of corresponding wide-character codes and shall store not more than `n` wide-character codes into the array pointed to by `pwcs`. No characters that follow a null byte (which is converted into a wide-character code with value 0) shall be examined or converted. Each character shall be converted as if by a call to `mbtowc()`, except that the shift state of `mbtowc()` is not affected.

No more than `n` elements shall be modified in the array pointed to by `pwcs`. If copying takes place between objects that overlap, the behavior is undefined.

The behavior of this function shall be affected by the `LC_CTYPE` category of the current locale. If `pwcs` is a null pointer, `mbstowcs()` shall return the length required to convert the entire array regardless of the value of `n`, but no values are stored.

**RETURN VALUE**

If an invalid character is encountered, `mbstowcs()` shall return `(size_t)-1` and may set `errno` to indicate the error.

Otherwise, `mbstowcs()` shall return the number of the array elements modified (or required if `pwcs` is null), not including a terminating 0 code, if any. The array shall not be zero-terminated if the value returned is `n`.

**ERRORS**

The `mbstowcs()` function may fail if:

- `[EILSEQ]` Invalid byte sequence is detected.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`mblen()`, `mbtowc()`, `wctomb()`, `wcstombs()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`
CHANGE HISTORY

First released in Issue 4. Aligned with the ISO C standard.

Issue 6

The `mbstowcs()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

Extensions beyond the ISO C standard are marked.
NAME
mbtowc — convert a character to a wide-character code

SYNOPSIS
#include <stdlib.h>

int mbtowc(wchar_t *restrict pwc, const char *restrict s, size_t n);

DESCRIPTION
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

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.

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.

The implementation shall behave as if no function defined in this volume of
IEEE Std 1003.1-2001 calls mbtowc().

RETURN VALUE
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).

In no case shall the value returned be greater than n or the value of the {MB_CUR_MAX} macro.

ERRORS
The mbtowc() function may fail if:

[EINVAL] Invalid character sequence is detected.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO

`mblen()`, `mbstowcs()`, `wctomb()`, `wcstombs()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`

CHANGE HISTORY

First released in Issue 4. Aligned with the ISO C standard.

Issue 6

The `mbtowc()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

Extensions beyond the ISO C standard are marked.
memccpy()

NAME
memccpy — copy bytes in memory

SYNOPSIS
XSI
#include <string.h>

void *memccpy(void *restrict s1, const void *restrict s2,
int c, size_t n);

DESCRIPTION
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.

RETURN VALUE
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.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
The memccpy() function does not check for the overflow of the receiving memory area.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The restrict keyword is added to the memccpy() prototype for alignment with the ISO/IEC 9899: 1999 standard.
NAME
 memchr — find byte in memory

SYNOPSIS
#include <string.h>

void *memchr(const void *s, int c, size_t n);

DESCRIPTION
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
The **memchr**() function shall locate the first occurrence of **c** (converted to an **unsigned char**) in
the initial **n** bytes (each interpreted as **unsigned char**) of the object pointed to by **s**.

RETURN VALUE
The **memchr**() function shall return a pointer to the located byte, or a null pointer if the byte does
not occur in the object.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME

 memcmp — compare bytes in memory

SYNOPSIS

#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);

DESCRIPTION

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 memcmp() function shall compare the first n bytes (each interpreted as unsigned char) of the object pointed to by s1 to the first n bytes of the object pointed to by s2.

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 bytes (both interpreted as type unsigned char) that differ in the objects being compared.

RETURN VALUE

The memcmp() function shall return an integer greater than, equal to, or less than 0, if the object pointed to by s1 is greater than, equal to, or less than the object pointed to by s2, respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
memcpy — copy bytes in memory

SYNOPSIS
#include <string.h>

void *memcpy(void *restrict s1, const void *restrict s2, size_t n);

DESCRIPTION
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 memcpy() function shall copy n bytes from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE
The memcpy() function shall return s1; no return value is reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
The memcpy() function does not check for the overflow of the receiving memory area.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The memcpy() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
memmove — copy bytes in memory with overlapping areas

SYNOPSIS
#include <string.h>

void *memmove(void *s1, const void *s2, size_t n);

DESCRIPTION
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 memmove() function shall copy n bytes from the object pointed to by s2 into the object pointed to by s1. Copying takes place as if the n bytes from the object pointed to by s2 are first copied into a temporary array of n bytes that does not overlap the objects pointed to by s1 and s2, and then the n bytes from the temporary array are copied into the object pointed to by s1.

RETURN VALUE
The memmove() function shall return s1; no return value is reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 4. Derived from the ANSI C standard.
NAME
memset — set bytes in memory

SYNOPSIS
#include <string.h>

void *memset(void *s, int c, size_t n);

DESCRIPTION
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 memset() function shall copy c (converted to an unsigned char) into each of the first n bytes of the object pointed to by s.

RETURN VALUE
The memset() function shall return s; no return value is reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
mkdir — make a directory

SYNOPSIS
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);

DESCRIPTION
The mkdir() function shall create a new directory with name path. The file permission bits of the
new directory shall be initialized from mode. These file permission bits of the mode argument
shall be modified by the process’ file creation mask.

When bits in mode other than the file permission bits are set, the meaning of these additional bits
is implementation-defined.

The directory’s user ID shall be set to the process’ effective user ID. The directory’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 directory’s group ID to the group ID of the
parent directory. Implementations may, but need not, provide an implementation-defined way
to initialize the directory’s group ID to the effective group ID of the calling process.

The newly created directory shall be an empty directory.

If path names a symbolic directory, mkdir() shall fail and set errno to [EEXIST].

Upon successful completion, mkdir() shall mark for update the st_atime, st_ctime, and st_mtime
fields of the directory. Also, the st_ctime and st_mtime fields of the directory that contains the
new entry shall be marked for update.

RETURN VALUE
Upon successful completion, mkdir() shall return 0. Otherwise, −1 shall be returned, no directory
shall be created, and errno shall be set to indicate the error.

ERRORS
The mkdir() function shall fail if:

[EACCES] Search permission is denied on a component of the path prefix, or write
permission is denied on the parent directory of the directory to be created.

[EEXIST] The named file exists.

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

[ENAMETOOLONG] The length of the path argument exceeds [PATH_MAX] or a pathname
component is longer than [NAME_MAX].

[ENOENT] A component of the path prefix specified by path does not name an existing
directory or path is an empty string.

[ENOSPC] The file system does not contain enough space to hold the contents of the new
directory or to extend the parent directory of the new directory.

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

[EROFS] The parent directory resides on a read-only file system.
The `mkdir()` function may fail if:

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

**[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}`.

**EXAMPLES**

**Creating a Directory**

The following example shows how to create a directory named `/home/cnd/mod1`, with read/write/search permissions for owner and group, and with read/search permissions for others.

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
...
status = mkdir("/home/cnd/mod1", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
```

**APPLICATION USAGE**

None.

**RATIONALE**

The `mkdir()` function originated in 4.2 BSD and was added to System V in Release 3.0.

4.3 BSD detects `[ENAMETOOLONG]`.

The POSIX.1-1990 standard required that the group ID of a newly created directory be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use `chown()` to set the group ID after the directory is created, or determine under what conditions the implementation will set the desired group ID.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`umask()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/stat.h>`, `<sys/types.h>`

**CHANGE HISTORY**

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

**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.
• The [ELOOP] mandatory error condition is added.

• A second [ENAMETOOLONG] is added as an optional error condition.

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

• The [ELOOP] optional error condition is added.
NAME
mkfifo — make a FIFO special file

SYNOPSIS
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);

DESCRIPTION
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.

When bits in mode other than the file permission bits are set, the effect is implementation-defined.

If path names a symbolic link, mkfifo() shall fail and set errno to [EEXIST].

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.

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.

RETURN VALUE
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.

ERRORS
The mkfifo() function shall fail if:

[EACCES] A component of the path prefix denies search permission, or write permission is denied on the parent directory of the FIFO to be created.

[EEXIST] The named file already exists.

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

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

[ENOENT] A component of the path prefix specified by path does not name an existing directory or path is an empty string.

[ENOSPC] The directory that would contain the new file cannot be extended or the file system is out of file-allocation resources.

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

[EROFS] The named file resides on a read-only file system.

The mkfifo() function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the path argument.
As a result of encountering a symbolic link in resolution of the path argument, the length of the substituted pathname string exceeded \{PATH_MAX\}.

**EXAMPLES**

**Creating a FIFO File**

The following example shows how to create a FIFO file named `/home/cnd/mod_done`, with read/write permissions for owner, and with read permissions for group and others.

```c
#include <sys/types.h>
#include <sys/stat.h>

int status;
...
status = mkfifo("/home/cnd/mod_done", S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH);
```

**APPLICATION USAGE**

None.

**RATIONALE**

The syntax of this function is intended to maintain compatibility with historical implementations of `mknod()` function. The latter function was included in the 1984 /usr/group standard but only for use in creating FIFO special files. The `mknod()` function was originally excluded from the POSIX.1-1988 standard as implementation-defined and replaced by `mkdir()` and `mkfifo()`. The `mknod()` function is now included for alignment with the Single UNIX Specification.

The POSIX.1-1990 standard required that the group ID of a newly created FIFO be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use `chown()` to set the group ID after the FIFO is created, or determine under what conditions the implementation will set the desired group ID.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`umask()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/stat.h>`, `<sys/types.h>`

**CHANGE HISTORY**

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

**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.
• The [ELOOP] mandatory error condition is added.

• A second [ENOMEMTOOLONG] is added as an optional error condition.

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

• The [ELOOP] optional error condition is added.
mknod()  System Interfaces

NAME
mknod — make a directory, a special file, or a regular file

SYNOPSIS
XSI
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, dev_t dev);

DESCRIPTION
The mknod() function shall create a new file named by the pathname to which the argument path points.
The file type for path is OR’ed into the mode argument, and the application shall select one of the following symbolic constants:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IFIFO</td>
<td>FIFO-special</td>
</tr>
<tr>
<td>S_IFCHR</td>
<td>Character-special (non-portable)</td>
</tr>
<tr>
<td>S_IFDIR</td>
<td>Directory (non-portable)</td>
</tr>
<tr>
<td>S_IFBLK</td>
<td>Block-special (non-portable)</td>
</tr>
<tr>
<td>S_IFREG</td>
<td>Regular (non-portable)</td>
</tr>
</tbody>
</table>

The only portable use of mknod() is to create a FIFO-special file. If mode is not S_IFIFO or dev is not 0, the behavior of mknod() is unspecified.
The permissions for the new file are OR’ed into the mode argument, and may be selected from any combination of the following symbolic constants:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_ISUID</td>
<td>Set user ID on execution.</td>
</tr>
<tr>
<td>S_ISGID</td>
<td>Set group ID on execution.</td>
</tr>
<tr>
<td>S_IRWXU</td>
<td>Read, write, or execute (search) by owner.</td>
</tr>
<tr>
<td>S_IRUSR</td>
<td>Read by owner.</td>
</tr>
<tr>
<td>S_IWUSR</td>
<td>Write by owner.</td>
</tr>
<tr>
<td>S_IWUSR</td>
<td>Write by owner.</td>
</tr>
<tr>
<td>S_IRUSR</td>
<td>Execute (search) by owner.</td>
</tr>
<tr>
<td>S_IWUSR</td>
<td>Read by group.</td>
</tr>
<tr>
<td>S_IRWXG</td>
<td>Read, write, or execute (search) by group.</td>
</tr>
<tr>
<td>S_IRGRP</td>
<td>Read by group.</td>
</tr>
<tr>
<td>S_IWGRP</td>
<td>Write by group.</td>
</tr>
<tr>
<td>S_IWGRP</td>
<td>Execute (search) by group.</td>
</tr>
<tr>
<td>S_IRWXO</td>
<td>Read, write, or execute (search) by others.</td>
</tr>
<tr>
<td>S_IROTH</td>
<td>Read by others.</td>
</tr>
<tr>
<td>S_IWOTH</td>
<td>Write by others.</td>
</tr>
<tr>
<td>S_IWOTH</td>
<td>Write by others.</td>
</tr>
<tr>
<td>S_IWOTH</td>
<td>Execute (search) by others.</td>
</tr>
<tr>
<td>S_ISVTX</td>
<td>On directories, restricted deletion flag.</td>
</tr>
</tbody>
</table>

The user ID of the file shall be initialized to the effective user ID of the process. The group ID of the file shall be initialized to either the effective group ID of the process or the group ID of the parent directory. 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. The owner, group, and other permission bits of mode shall be modified by the file mode creation mask of the process. The mknod() function shall clear each bit whose corresponding bit in the file mode creation mask of the process is set.
If *path* names a symbolic link, `mknod()` shall fail and set `errno` to [EEXIST].

Upon successful completion, `mknod()` 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.

Only a process with appropriate privileges may invoke `mknod()` for file types other than FIFO-special.

**RETURN VALUE**

Upon successful completion, `mknod()` shall return 0. Otherwise, it shall return −1, the new file shall not be created, and `errno` shall be set to indicate the error.

**ERRORS**

The `mknod()` function shall fail if:

- [EACCES] A component of the path prefix denies search permission, or write permission is denied on the parent directory.
- [EEXIST] The named file exists.
- [EINVAL] An invalid argument exists.
- [EIO] An I/O error occurred while accessing the file system.
- [ELOOP] A loop exists in symbolic links encountered during resolution of the `path` argument.
- [ENAMETOOLONG] The length of a pathname exceeds [PATH_MAX] or a pathname component is longer than [NAME_MAX].
- [ENOENT] A component of the path prefix specified by `path` does not name an existing directory or `path` is an empty string.
- [ENOSPC] The directory that would contain the new file cannot be extended or the file system is out of file allocation resources.
- [ENOTDIR] A component of the path prefix is not a directory.
- [EPERM] The invoking process does not have appropriate privileges and the file type is not FIFO-special.
- [EROFS] The directory in which the file is to be created is located on a read-only file system.

The `mknod()` function may fail if:

- [ELOOP] More than [SYMLOOP_MAX] symbolic links were encountered during resolution of the `path` argument.
- [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds [PATH_MAX].
EXAMPLES

Creating a FIFO Special File

The following example shows how to create a FIFO special file named /home/cnd/mod_done, with read/write permissions for owner, and with read permissions for group and others.

```c
#include <sys/types.h>
#include <sys/stat.h>

dev_t dev;
int status;
...
status = mknod("/home/cnd/mod_done", S_IFIFO | S_IWUSR |
              S_IRUSR | S_IRGRP | S_IROTH, dev);
```

APPLICATION USAGE

The mkfifo() function is preferred over this function for making FIFO special files.

RATIONALE

The POSIX.1-1990 standard required that the group ID of a newly created file be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use chown() to set the group ID after the file is created, or determine under what conditions the implementation will set the desired group ID.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod(), creat(), exec(), mkdir(), mkfifo(), open(), stat(), umask(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/stat.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

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

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.
NAME
mkstemp — make a unique filename

SYNOPSIS
#include <stdlib.h>

int mkstemp(char *template);

DESCRIPTION
The mkstemp() function shall replace the contents of the string pointed to by template by a unique
filename, and return a file descriptor for the file open for reading and writing. The function thus
prevents any possible race condition between testing whether the file exists and opening it for
use. The string in template should look like a filename with six trailing 'X' s; mkstemp() replaces
each 'X' with a character from the portable filename character set. The characters are chosen
such that the resulting name does not duplicate the name of an existing file at the time of a call
to mkstemp().

RETURN VALUE
Upon successful completion, mkstemp() shall return an open file descriptor. Otherwise, −1 shall
be returned if no suitable file could be created.

ERRORS
No errors are defined.

EXAMPLES
Generating a Filename
The following example creates a file with a 10-character name beginning with the characters
"file" and opens the file for reading and writing. The value returned as the value of fd is a file
descriptor that identifies the file.

#include <stdlib.h>
...
char template[] = "/tmp/fileXXXXXX";
int fd;
fd = mkstemp(template);

APPLICATION USAGE
It is possible to run out of letters.

The mkstemp() function need not check to determine whether the filename part of template
exceeds the maximum allowable filename length.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getpid(), open(), tmpfile(), tmpnam(), the Base Definitions volume of IEEE Std 1003.1-2001,
<stdlib.h>
mkstemp()

CHANGE HISTORY
First released in Issue 4, Version 2.
Moved from X/OPEN UNIX extension to BASE.

Issue 5
NAME
mktemp — make a unique filename (LEGACY)

SYNOPSIS
XSI
#include <stdlib.h>
char *mktemp(char *template);

DESCRIPTION
The mktemp() function shall replace the contents of the string pointed to by template by a unique
filename and return template. The application shall initialize template to be a filename with six
trailing 'X's; mktemp() shall replace each 'X' with a single byte character from the portable
filename character set.

RETURN VALUE
The mktemp() function shall return the pointer template. If a unique name cannot be created,
template shall point to a null string.

ERRORS
No errors are defined.

EXAMPLES
Generating a Filename
The following example replaces the contents of the "template" string with a 10-character
filename beginning with the characters "file" and returns a pointer to the "template" string
that contains the new filename.

include <stdlib.h>
...
char *template = "/tmp/fileXXXXXX";
char *ptr;
ptr = mktemp(template);

APPLICATION USAGE
Between the time a pathname is created and the file opened, it is possible for some other process
to create a file with the same name. The mkstemp() function avoids this problem and is preferred
over this function.

RATIONALE
None.

FUTURE DIRECTIONS
This function may be withdrawn in a future version.

SEE ALSO
mkstemp(), tmpfile(), tmpnam(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
This function is marked LEGACY.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
mktime — convert broken-down time into time since the Epoch

SYNOPSIS
#include <time.h>
time_t mktime(struct tm *timeptr);

DESCRIPTION
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

The mktime() function shall convert the broken-down time, expressed as local time, in the
structure pointed to by timeptr, into a time since the Epoch value with the same encoding as that
of the values returned by time(). The original values of the tm_wday and tm_yday components of
the structure are ignored, and the original values of the other components are not restricted to
the ranges described in <time.h>.

A positive or 0 value for tm_isdst shall cause mktime() to presume initially that Daylight Savings
Time, respectively, is or is not in effect for the specified time. A negative value for tm_isdst shall
cause mktime() to attempt to determine whether Daylight Savings Time is in effect for the
specified time.

Local timezone information shall be set as though mktime() called tzset().

The relationship between the tm structure (defined in the <time.h> header) and the time in
seconds since the Epoch 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) corrected for timezone and any seasonal time
adjustments, where the names in the structure and in the expression correspond.

Upon successful completion, the values of the tm_wday and tm_yday components of the structure
shall be set appropriately, and the other components are set to represent the specified time since
the Epoch, but with their values forced to the ranges indicated in the <time.h> entry; the final
value of tm_mday shall not be set until tm_mon and tm_year are determined.

RETURN VALUE
The mktime() function shall return the specified time since the Epoch encoded as a value of type
time_t. If the time since the Epoch cannot be represented, the function shall return the value
(time_t)-1.

ERRORS
No errors are defined.

EXAMPLES
What day of the week is July 4, 2001?
#include <stdio.h>
#include <time.h>
struct tm time_str;
char daybuf[20];
int main(void)
{
    time_str.tm_year = 2001 - 1900;
    time_str.tm_mon = 7 - 1;
    time_str.tm_mday = 4;
```c
#include <time.h>

time_t mktime(struct tm *time_str)
{
    time_str->tm_hour = 0;
    time_str->tm_min = 0;
    time_str->tm_sec = 1;
    time_str->tm_isdst = -1;
    if (mktime(&time_str) == -1)
        (void)puts("-unknown-"),
    else {
        (void)strftime(datebuf, sizeof(datebuf), "%A", &time_str);
        (void)puts(datebuf);
    }
    return 0;
}
```

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
asctime, clock, ctime, difftime, gmtime, localtime, strftime, strptime, time, utime,
the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

**CHANGE HISTORY**

**Issue 6**
Extensions beyond the ISO C standard are marked.
NAME
mlock, munlock — lock or unlock a range of process address space (REALTIME)

SYNOPSIS
#include <sys/mman.h>

int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);

DESCRIPTION
The mlock() function shall cause those whole pages containing any part of the address space of
the process starting at address addr and continuing for len bytes to be memory-resident until
unlocked or until the process exits or execs another process image. The implementation may
require that addr be a multiple of {PAGESIZE}.

The munlock() function shall unlock those whole pages containing any part of the address space
of the process starting at address addr and continuing for len bytes, regardless of how many
times mlock() has been called by the process for any of the pages in the specified range. The
implementation may require that addr be a multiple of {PAGESIZE}.

If any of the pages in the range specified to a call to munlock() are also mapped into the address
spaces of other processes, any locks established on those pages by another process are
unaffected by the call of this process to munlock(). If any of the pages in the range specified by a
call to munlock() are also mapped into other portions of the address space of the calling process
outside the range specified, any locks established on those pages via the other mappings are also
unaffected by this call.

Upon successful return from mlock(), pages in the specified range shall be locked and memory-
resident. Upon successful return from munlock(), pages in the specified range shall be unlocked
with respect to the address space of the process. Memory residency of unlocked pages is
unspecified.

The appropriate privilege is required to lock process memory with mlock().

RETURN VALUE
Upon successful completion, the mlock() and munlock() functions shall return a value of zero.
Otherwise, no change is made to any locks in the address space of the process, and the function
shall return a value of −1 and set errno to indicate the error.

ERRORS
The mlock() and munlock() functions shall fail if:

[ENOMEM] Some or all of the address range specified by the addr and len arguments does
not correspond to valid mapped pages in the address space of the process.

The mlock() function shall fail if:

[EAGAIN] Some or all of the memory identified by the operation could not be locked
when the call was made.

The mlock() and munlock() functions may fail if:

[EINVAL] The addr argument is not a multiple of {PAGESIZE}.

The mlock() function may fail if:

[ENOMEM] Locking the pages mapped by the specified range would exceed an
implementation-defined limit on the amount of memory that the process may
lock.
mlock()

The calling process does not have the appropriate privilege to perform the requested operation.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
exec, exit(), fork(), mlockall(), munmap(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/mman.h>

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

Issue 6
The mlock() and munlock() functions are marked as part of the Range Memory Locking option.
The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Range Memory Locking option.
mlockall( )

NAME
mlockall, munlockall — lock/unlock the address space of a process (REALTIME)

SYNOPSIS
#include <sys/mman.h>
int mlockall(int flags);
int munlockall(void);

DESCRIPTION
The mlockall() function shall cause all of the pages mapped by the address space of a process to be memory-resident until unlocked or until the process exits or execs another process image. The flags argument determines whether the pages to be locked are those currently mapped by the address space of the process, those that are mapped in the future, or both. The flags argument is constructed from the bitwise-inclusive OR of one or more of the following symbolic constants, defined in <sys/mman.h>:

MCL_CURRENT Lock all of the pages currently mapped into the address space of the process.
MCL_FUTURE Lock all of the pages that become mapped into the address space of the process in the future, when those mappings are established.

If MCL_FUTURE is specified, and the automatic locking of future mappings eventually causes the amount of locked memory to exceed the amount of available physical memory or any other implementation-defined limit, the behavior is implementation-defined. The manner in which the implementation informs the application of these situations is also implementation-defined.

The munlockall() function shall unlock all currently mapped pages of the address space of the process. Any pages that become mapped into the address space of the process after a call to munlockall() shall not be locked, unless there is an intervening call to mlockall() specifying MCL_FUTURE or a subsequent call to mlockall() specifying MCL_CURRENT. If pages mapped into the address space of the 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 a call by this process to munlockall().

Upon successful return from the mlockall() function that specifies MCL_CURRENT, all currently mapped pages of the process’ address space shall be memory-resident and locked. Upon return from the munlockall() function, all currently mapped pages of the process’ address space shall be unlocked with respect to the process’ address space. The memory residency of unlocked pages is unspecified.

The appropriate privilege is required to lock process memory with mlockall().

RETURN VALUE
Upon successful completion, the mlockall() function shall return a value of zero. Otherwise, no additional memory shall be locked, and the function shall return a value of −1 and set errno to indicate the error. The effect of failure of mlockall() on previously existing locks in the address space is unspecified.

If it is supported by the implementation, the munlockall() function shall always return a value of zero. Otherwise, the function shall return a value of −1 and set errno to indicate the error.

ERRORS
The mlockall() function shall fail if:

[EAGAIN] Some or all of the memory identified by the operation could not be locked when the call was made.
mlockall()

The flags argument is zero, or includes unimplemented flags.

The mlockall() function may fail if:

[EINVAL] The flags argument is zero, or includes unimplemented flags.

[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.

[EPERM] The calling process does not have the appropriate privilege to perform the requested operation.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
exec, exit(), fork(), mlock(), munmap(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/mman.h>

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

Issue 6
The mlockall() and munlockall() functions are marked as part of the Process Memory Locking option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Memory Locking option.
NAME
mmap — map pages of memory

SYNOPSIS

```c
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags,
            int fildes, off_t off);
```

DESCRIPTION

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:

```c
pa=mmap(addr, len, prot, flags, fildes, off);
```

The `mmap()` function shall establish a mapping between the address space of the process at an address `pa` for `len` bytes to the memory object represented by the file descriptor `fildes` at offset `off` for `len` bytes. The value of `pa` is an implementation-defined function of the parameter `addr` and the values of `flags`, further described below. A successful `mmap()` call shall return `pa` as its result.

The address range starting at `pa` and continuing for `len` bytes shall be legitimate for the possible (not necessarily current) address space of the process. The range of bytes starting at `off` and continuing for `len` bytes shall be legitimate for the possible (not necessarily current) offsets in the file, shared memory object, or typed memory object represented by `fildes`.

If `fildes` represents a typed memory object opened with either the `POSIX_TYPED_MEM_ALLOCATE` flag or the `POSIX_TYPED_MEM_ALLOCATE_CONTIG` flag, the memory object to be mapped shall be that portion of the typed memory object allocated by the implementation as specified below. In this case, if `off` is non-zero, the behavior of `mmap()` is undefined. If `fildes` refers to a valid typed memory object that is not accessible from the calling process, `mmap()` shall fail.

The mapping established by `mmap()` shall replace any previous mappings for those whole pages containing any part of the address space of the process starting at `pa` and continuing for `len` bytes.

If the size of the mapped file changes after the call to `mmap()` as a result of some other operation on the mapped file, the effect of references to portions of the mapped region that correspond to added or removed portions of the file is unspecified.

The `mmap()` function shall be supported for regular files, shared memory objects, and typed memory objects. Support for any other type of file is unspecified.

The parameter `prot` determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The `prot` shall be either `PROT_NONE` or the bitwise-inclusive OR of one or more of the other flags in the following table, defined in the `<sys/mman.h>` header.

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROT_READ</td>
<td>Data can be read.</td>
</tr>
<tr>
<td>PROT_WRITE</td>
<td>Data can be written.</td>
</tr>
<tr>
<td>PROT_EXEC</td>
<td>Data can be executed.</td>
</tr>
<tr>
<td>PROT_NONE</td>
<td>Data cannot be accessed.</td>
</tr>
</tbody>
</table>

If an implementation cannot support the combination of access types specified by `prot`, the call to `mmap()` shall fail.
An implementation may permit accesses other than those specified by prot; however, if the Memory Protection option is supported, the implementation shall not permit a write to succeed where PROT_WRITE has not been set or shall not permit any access where PROT_NONE alone has been set. The implementation shall support at least the following values of prot: PROT_NONE, PROT_READ, PROT_WRITE, and the bitwise-inclusive OR of PROT_READ and PROT_WRITE. If the Memory Protection option is not supported, the result of any access that conflicts with the specified protection is undefined. The file descriptor fildes shall have been opened with read permission, regardless of the protection options specified. If PROT_WRITE is specified, the application shall ensure that it has opened the file descriptor fildes with write permission unless MAP_PRIVATE is specified in the flags parameter as described below.

The parameter flags provides other information about the handling of the mapped data. The value of flags is the bitwise-inclusive OR of these options, defined in <sys/mman.h>:

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP_SHARED</td>
<td>Changes are shared.</td>
</tr>
<tr>
<td>MAP_PRIVATE</td>
<td>Changes are private.</td>
</tr>
<tr>
<td>MAP_FIXED</td>
<td>Interpret addr exactly.</td>
</tr>
</tbody>
</table>

Implementations that do not support the Memory Mapped Files option are not required to support MAP_PRIVATE.

It is implementation-defined whether MAP_FIXED shall be supported. MAP_FIXED shall be supported on XSI-conformant systems.

MAP_SHARED and MAP_PRIVATE describe the disposition of write references to the memory object. If MAP_SHARED is specified, write references shall change the underlying object. If MAP_PRIVATE is specified, modifications to the mapped data by the calling process shall be visible only to the calling process and shall not change the underlying object. It is unspecified whether modifications to the underlying object done after the MAP_PRIVATE mapping is established are visible through the MAP_PRIVATE mapping. Either MAP_SHARED or MAP_PRIVATE can be specified, but not both. The mapping type is retained across fork().

When fildes represents a typed memory object opened with either the POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, mmap() shall, if there are enough resources available, map len bytes allocated from the corresponding typed memory object which were not previously allocated to any process in any processor that may access that typed memory object. If there are not enough resources available, the function shall fail. If fildes represents a typed memory object opened with the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, these allocated bytes shall be contiguous within the typed memory object. If fildes represents a typed memory object opened with the POSIX_TYPED_MEM_ALLOCATE flag, these allocated bytes may be composed of non-contiguous fragments within the typed memory object. If fildes represents a typed memory object opened with neither the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag nor the POSIX_TYPED_MEM_ALLOCATE flag, len bytes starting at offset off within the typed memory object are mapped, exactly as when mapping a file or shared memory object. In this case, if two processes map an area of typed memory using the same off and len values and using file descriptors that refer to the same memory pool (either from the same port or from a different port), both processes shall map the same region of storage.

When MAP_FIXED is set in the flags argument, the implementation is informed that the value of pa shall be addr, exactly. If MAP_FIXED is set, mmap() may return MAP_FAILED and set errno to [EINVAL]. If a MAP_FIXED request is successful, the mapping established by mmap() replaces any previous mappings for the process’ pages in the range (pa, pa+len).
When MAP_FIXED is not set, the implementation uses addr in an implementation-defined manner to arrive at pa. The pa so chosen shall be an area of the address space that the implementation deems suitable for a mapping of len bytes to the file. All implementations interpret an addr value of 0 as granting the implementation complete freedom in selecting pa, subject to constraints described below. A non-zero value of addr is taken to be a suggestion of a process address near which the mapping should be placed. When the implementation selects a value for pa, it never places a mapping at address 0, nor does it replace any extant mapping.

The off argument is constrained to be aligned and sized according to the value returned by sysconf() when passed _SC_PAGESIZE or _SC_PAGE_SIZE. When MAP_FIXED is specified, the application shall ensure that the argument addr also meets these constraints. The implementation performs mapping operations over whole pages. Thus, while the argument len need not meet a size or alignment constraint, the implementation shall include, in any mapping operation, any partial page specified by the range [pa, pa+len).

The system shall always zero-fill any partial page at the end of an object. Further, the system shall never write out any modified portions of the last page of an object which are beyond its end. References within the address range starting at pa and continuing for len bytes to whole pages following the end of an object shall result in delivery of a SIGBUS signal.

An implementation may generate SIGBUS signals when a reference would cause an error in the mapped object, such as out-of-space condition.

The mmap() function shall add an extra reference to the file associated with the file descriptor fildes which is not removed by a subsequent close() on that file descriptor. This reference shall be removed when there are no more mappings to the file.

The st_atime field of the mapped file may be marked for update at any time between the mmap() call and the corresponding munmap() call. The initial read or write reference to a mapped region shall cause the file's st_atime field to be marked for update if it has not already been marked for update.

The st_ctime and st_mtime fields of a file that is mapped with MAP_SHARED and PROT_WRITE shall be marked for update at some point in the interval between a write reference to the mapped region and the next call to msync() with MS_ASYNC or MS_SYNC for that portion of the file by any process. If there is no such call and if the underlying file is modified as a result of a write reference, then these fields shall be marked for update at some time after the write reference.

There may be implementation-defined limits on the number of memory regions that can be mapped (per process or per system).

If such a limit is imposed, whether the number of memory regions that can be mapped by a process is decreased by the use of shmat() is implementation-defined.

If mmap() fails for reasons other than [EBADF], [EINVAL], or [ENOTSUP], some of the mappings in the address range starting at addr and continuing for len bytes may have been unmapped.

**RETURN VALUE**

Upon successful completion, the mmap() function shall return the address at which the mapping was placed (pa); otherwise, it shall return a value of MAP_FAILED and set errno to indicate the error. The symbol MAP_FAILED is defined in the <sys/mman.h> header. No successful return from mmap() shall return the value MAP_FAILED.
The *mmap()* function shall fail if:

- **[EACCES]** The *fildes* argument is not open for read, regardless of the protection specified, or *fildes* is not open for write and PROT_WRITE was specified for a MAP_SHARED type mapping.
- **[EAGAIN]** The mapping could not be locked in memory, if required by *mlockall()*, due to a lack of resources.
- **[EBADF]** The *fildes* argument is not a valid open file descriptor.
- **[EINVAL]** The *addr* argument (if MAP_FIXED was specified) or *off* is not a multiple of the page size as returned by *sysconf()*, or is considered invalid by the implementation.
- **[EINVAL]** The value of *flags* is invalid (neither MAP_PRIVATE nor MAP_SHARED is set).
- **[EMFILE]** The number of mapped regions would exceed an implementation-defined limit (per process or per system).
- **[ENODEV]** The *fildes* argument refers to a file whose type is not supported by *mmap()*.  
- **[ENOMEM]** MAP_FIXED was specified, and the range *[addr, addr+len]* exceeds that allowed for the address space of a process; or, if MAP_FIXED was not specified and there is insufficient room in the address space to effect the mapping.
- **[ENOMEM]** The mapping could not be locked in memory, if required by *mlockall()*, because it would require more space than the system is able to supply.
- **[ENOMEM]** Not enough unallocated memory resources remain in the typed memory object designated by *fildes* to allocate *len* bytes.
- **[ENOTSUP]** MAP_FIXED or MAP_PRIVATE was specified in the *flags* argument and the implementation does not support this functionality.
- **[ENXIO]** Addresses in the range *[off, off+len]* are invalid for the object specified by *fildes*.
- **[ENXIO]** MAP_FIXED was specified in *flags* and the combination of *addr*, *len*, and *off* is invalid for the object specified by *fildes*.
- **[ENXIO]** The *fildes* argument refers to a typed memory object that is not accessible from the calling process.
- **[EOVERFLOW]** The file is a regular file and the value of *off* plus *len* exceeds the offset maximum established in the open file description associated with *fildes*.

**Examples**

None.

**Application Usage**

Use of *mmap()* may reduce the amount of memory available to other memory allocation functions.

Use of MAP_FIXED may result in unspecified behavior in further use of *malloc()* and *shmat()*.

The use of MAP_FIXED is discouraged, as it may prevent an implementation from making the most effective use of resources.
The application must ensure correct synchronization when using `mmap()` in conjunction with any other file access method, such as `read()` and `write()`, standard input/output, and `shmat()`.

The `mmap()` function allows access to resources via address space manipulations, instead of `read()`/`write()`. Once a file is mapped, all a process has to do to access it is use the data at the address to which the file was mapped. So, using pseudo-code to illustrate the way in which an existing program might be changed to use `mmap()`, the following:

```c
fildes = open(...)  
llseek(fildes, some_offset)  
read(fildes, buf, len)  
/* Use data in buf. */
```

becomes:

```c
fildes = open(...)  
address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)  
/* Use data at address. */
```

Rationale

After considering several other alternatives, it was decided to adopt the `mmap()` definition found in SVR4 for mapping memory objects into process address spaces. The SVR4 definition is minimal, in that it describes only what has been built, and what appears to be necessary for a general and portable mapping facility.

Note that while `mmap()` was first designed for mapping files, it is actually a general-purpose mapping facility. It can be used to map any appropriate object, such as memory, files, devices, and so on, into the address space of a process.

When a mapping is established, it is possible that the implementation may need to map more than is requested into the address space of the process because of hardware requirements. An application, however, cannot count on this behavior. Implementations that do not use a paged architecture may simply allocate a common memory region and return the address of it; such implementations probably do not allocate any more than is necessary. References past the end of the requested area are unspecified.

If an application requests a mapping that would overlay existing mappings in the process, it might be desirable that an implementation detect this and inform the application. However, the default, portable (not MAP_FIXED) operation does not overlay existing mappings. On the other hand, if the program specifies a fixed address mapping (which requires some implementation knowledge to determine a suitable address, if the function is supported at all), then the program is presumed to be successfully managing its own address space and should be trusted when it asks to map over existing data structures. Furthermore, it is also desirable to make as few system calls as possible, and it might be considered onerous to require an `munmap()` before an `mmap()` to the same address range. This volume of IEEE Std 1003.1-2001 specifies that the new mappings replace any existing mappings, following existing practice in this regard.

It is not expected, when the Memory Protection option is supported, that all hardware implementations are able to support all combinations of permissions at all addresses. When this option is supported, implementations are required to disallow write access to mappings without write permission and to disallow access to mappings without any access permission. Other than these restrictions, implementations may allow access types other than those requested by the application. For example, if the application requests only PROT_WRITE, the implementation may also allow read access. A call to `mmap()` fails if the implementation cannot support allowing all the access requested by the application. For example, some implementations cannot support a request for both write access and execute access simultaneously. All implementations supporting the Memory Protection option must support requests for no access, read access,
write access, and both read and write access. Strictly conforming code must only rely on the
required checks. These restrictions allow for portability across a wide range of hardware.

The MAP_FIXED address treatment is likely to fail for non-page-aligned values and for certain
architecture-dependent address ranges. Conforming implementations cannot count on being
able to choose address values for MAP_FIXED without utilizing non-portable, implementation-
defined knowledge. Nonetheless, MAP_FIXED is provided as a standard interface conforming to
existing practice for utilizing such knowledge when it is available.

Similarly, in order to allow implementations that do not support virtual addresses, support for
directly specifying any mapping addresses via MAP_FIXED is not required and thus a
conforming application may not count on it.

The MAP_PRIVATE function can be implemented efficiently when memory protection hardware
is available. When such hardware is not available, implementations can implement such
“mappings” by simply making a real copy of the relevant data into process private memory,
though this tends to behave similarly to read().

The function has been defined to allow for many different models of using shared memory.
However, all uses are not equally portable across all machine architectures. In particular, the
mmap() function allows the system as well as the application to specify the address at which to
map a specific region of a memory object. The most portable way to use the function is always to
let the system choose the address, specifying NULL as the value for the argument addr and not
to specify MAP_FIXED.

If it is intended that a particular region of a memory object be mapped at the same address in a
group of processes (on machines where this is even possible), then MAP_FIXED can be used to
pass the desired mapping address. The system can still be used to choose the desired address
if the first such mapping is made without specifying MAP_FIXED, and then the resulting
mapping address can be passed to subsequent processes for them to pass in via MAP_FIXED.
The availability of a specific address range cannot be guaranteed, in general.

The mmap() function can be used to map a region of memory that is larger than the current size
of the object. Memory access within the mapping but beyond the current end of the underlying
objects may result in SIGBUS signals being sent to the process. The reason for this is that the size
of the object can be manipulated by other processes and can change at any moment. The
implementation should tell the application that a memory reference is outside the object where
this can be detected; otherwise, written data may be lost and read data may not reflect actual
data in the object.

Note that references beyond the end of the object do not extend the object as the new end cannot
be determined precisely by most virtual memory hardware. Instead, the size can be directly
manipulated by ftruncate().

Process memory locking does apply to shared memory regions, and the MEMLOCK_FUTURE
argument to mlockall() can be relied upon to cause new shared memory regions to be
automatically locked.

Existing implementations of mmap() return the value −1 when unsuccessful. Since the casting of
this value to type void * cannot be guaranteed by the ISO C standard to be distinct from a
successful value, this volume of IEEE Std 1003.1-2001 defines the symbol MAP_FAILED, which a
conforming implementation does not return as the result of a successful call.

FUTURE DIRECTIONS

None.
mmap()

SEE ALSO
exec, fcntl(), fork(), lockf(), msync(), munmap(), mprotect(), posix_typed_mem_open(), shmat(), sysconf(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/mman.h>

CHANGE HISTORY

Issue 5
First released in Issue 4, Version 2.
Moved from X/OPEN UNIX extension to BASE.
Aligned with mmap() in the POSIX Realtime Extension as follows:
• The DESCRIPTION is extensively reworded.
• The [EAGAIN] and [ENOTSUP] mandatory error conditions are added.
• New cases of [ENOMEM] and [ENXIO] are added as mandatory error conditions.
• The value returned on failure is the value of the constant MAP_FAILED; this was previously defined as −1.
Large File Summit extensions are added.

Issue 6
The mmap() function is marked as part of the Memory Mapped Files option.
The Open Group Corrigendum U028/6 is applied, changing (void *)−1 to MAP_FAILED.
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:
• The DESCRIPTION is updated to described the use of MAP_FIXED.
• The DESCRIPTION is updated to describe the addition of an extra reference to the file associated with the file descriptor passed to mmap().
• The DESCRIPTION is updated to state that there may be implementation-defined limits on the number of memory regions that can be mapped.
• The DESCRIPTION is updated to describe constraints on the alignment and size of the off argument.
• The [EINVAL] and [EMFILE] error conditions are added.
• The [EOVERFLOW] error condition is added. This change is to support large files.
The following changes are made for alignment with the ISO POSIX-1: 1996 standard:
• The DESCRIPTION is updated to describe the cases when MAP_PRIVATE and MAP_FIXED need not be supported.
The following changes are made for alignment with IEEE Std 1003.1j-2000:
• Semantics for typed memory objects are added to the DESCRIPTION.
• New [ENOMEM] and [ENXIO] errors are added to the ERRORS section.
• The posix_typed_mem_open() function is added to the SEE ALSO section.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
modf, modff, modfl — decompose a floating-point number

SYNOPSIS
#include <math.h>

double modf(double x, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);

DESCRIPTION
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

These functions shall break the argument x into integral and fractional parts, each of which has
the same sign as the argument. It stores the integral part as a double (for the modf() function), a
float (for the modff() function), or a long double (for the modfl() function), in the object pointed
to by iptr.

RETURN VALUE
Upon successful completion, these functions shall return the signed fractional part of x.

MX If x is NaN, a NaN shall be returned, and *iptr shall be set to a NaN.
If x is ±Inf, ±0 shall be returned, and *iptr shall be set to ±Inf.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
The modf() function computes the function result and *iptr such that:
a = modf(x, &iptr);
x == a + *iptr;
allowing for the usual floating-point inaccuracies.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
frexp(), isnan(), ldexp(), the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This
text was previously published in the APPLICATION USAGE section.

Issue 6
The modff() and modfl() functions are added for alignment with the ISO/IEC 9899:1999
standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

mprotect()

NAME
mprotect — set protection of memory mapping

SYNOPSIS

```
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

DESCRIPTION

The `mprotect()` function shall change the access protections to be that specified by `prot` for those whole pages containing any part of the address space of the process starting at address `addr` and continuing for `len` bytes. The parameter `prot` determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The `prot` argument should be either `PROT_NONE` or the bitwise-inclusive OR of one or more of `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`.

If an implementation cannot support the combination of access types specified by `prot`, the call to `mprotect()` shall fail.

An implementation may permit accesses other than those specified by `prot`; however, no implementation shall permit a write to succeed where `PROT_WRITE` has not been set or shall permit any access where `PROT_NONE` alone has been set. Implementations shall support at least the following values of `prot`: `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, and the bitwise-inclusive OR of `PROT_READ` and `PROT_WRITE`. If `PROT_WRITE` is specified, the application shall ensure that it has opened the mapped objects in the specified address range with write permission, unless `MAP_PRIVATE` was specified in the original mapping, regardless of whether the file descriptors used to map the objects have since been closed.

The implementation shall require that `addr` be a multiple of the page size as returned by `sysconf()`.

The behavior of this function is unspecified if the mapping was not established by a call to `mmap()`.

When `mprotect()` fails for reasons other than [EINVAL], the protections on some of the pages in the range `[addr, addr+len)` may have been changed.

RETURN VALUE

Upon successful completion, `mprotect()` shall return 0; otherwise, it shall return −1 and set `errno` to indicate the error.

ERRORS

The `mprotect()` function shall fail if:

- [EACCES] The `prot` argument specifies a protection that violates the access permission the process has to the underlying memory object.
- [EAGAIN] The `prot` argument specifies `PROT_WRITE` over a `MAP_PRIVATE` mapping and there are insufficient memory resources to reserve for locking the private page.
- [EINVAL] The `addr` argument is not a multiple of the page size as returned by `sysconf()`.
- [ENOMEM] Addresses in the range `[addr, addr+len)` are invalid for the address space of a process, or specify one or more pages which are not mapped.
- [ENOMEM] The `prot` argument specifies `PROT_WRITE` on a `MAP_PRIVATE` mapping, and it would require more space than the system is able to supply for locking the private pages, if required.
System Interfaces

mprotect()

25571 [ENOTSUP] The implementation does not support the combination of accesses requested in the prot argument.

25573 EXAMPLES
25574 None.

25575 APPLICATION USAGE
25576 The [EINVAL] error above is marked EX because it is defined as an optional error in the POSIX Realtime Extension.

25578 RATIONALE
25579 None.

25580 FUTURE DIRECTIONS
25581 None.

25582 SEE ALSO
25583 mmap(), sysconf(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/mman.h>

25584 CHANGE HISTORY
25585 First released in Issue 4, Version 2.

25586 Issue 5
25587 Moved from X/OPEN UNIX extension to BASE.
25588 Aligned with mprotect() in the POSIX Realtime Extension as follows:
25589 • The DESCRIPTION is largely reworded.
25590 • [ENOTSUP] and a second form of [ENOMEM] are added as mandatory error conditions.
25591 • [EAGAIN] is moved from the optional to the mandatory error conditions.

25592 Issue 6
25593 The mprotect() function is marked as part of the Memory Protection option.
25594 The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:
25595 • The DESCRIPTION is updated to state that implementations require addr to be a multiple of the page size as returned by sysconf().
25596 • The [EINVAL] error condition is added.
25597 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
mq_close()

NAME
mq_close — close a message queue (REALTIME)

SYNOPSIS
#include <mqueue.h>

int mq_close(mqd_t mqdes);

DESCRIPTION
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.

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.

RETURN VALUE
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.

ERRORS
The mq_close() function shall fail if:
[EBADF] The mqdes argument is not a valid message queue descriptor.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
mq_open(), mq_unlink(), msgctl(), msgget(), msgsend(), msgrcv(), msgsnd(), the Base Definitions volume of IEEE Std 1003.1-2001, <mqueue.h>

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

Issue 6
The mq_close() function is marked as part of the Message Passing option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.
NAME
mq_getattr — get message queue attributes (REALTIME)

SYNOPSIS
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);

DESCRIPTION
The mq_getattr() function shall obtain status information and attributes of the message queue
and the open message queue description associated with the message queue descriptor.
The mqdes argument specifies a message queue descriptor.
The results shall be returned in the mq_attr structure referenced by the mqstat argument.
Upon return, the following members shall have the values associated with the open message
queue description as set when the message queue was opened and as modified by subsequent
mq_setattr() calls: mq_flags.
The following attributes of the message queue shall be returned as set at message queue
creation: mq_maxmsg, mq_msgsize.
Upon return, the following members within the mq_attr structure referenced by the mqstat
argument shall be set to the current state of the message queue:
mq_curmsgs  The number of messages currently on the queue.

RETURN VALUE
Upon successful completion, the mq_getattr() function shall return zero. Otherwise, the function
shall return −1 and set errno to indicate the error.

ERRORS
The mq_getattr() function shall fail if:
[EBADF]  The mqdes argument is not a valid message queue descriptor.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
mq_open(), mq_send(), mq_setattr(), mq_timedsend(), msgctl(), msgget(), msgrcv(), msgsnd(), the
Base Definitions volume of IEEE Std 1003.1-2001, <mqueue.h>

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

Issue 6
The mq_getattr() function is marked as part of the Message Passing option.
The [ENOSYS] error condition has been removed as stubs need not be provided if an
implementation does not support the Message Passing option.
The `mq_timedsend()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
NAME
mq_notify — notify process that a message is available (REALTIME)

SYNOPSIS
#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent *notification);

DESCRIPTION
If the argument notification is not NULL, this function shall register the calling process to be
notified of message arrival at an empty message queue associated with the specified message
queue descriptor, mqdes. The notification specified by the notification argument shall be sent to
the process when the message queue transitions from empty to non-empty. At any time, only
one process may be registered for notification by a message queue. If the calling process or any
other process has already registered for notification of message arrival at the specified message
queue, subsequent attempts to register for that message queue shall fail.

If notification is NULL and the process is currently registered for notification by the specified
message queue, the existing registration shall be removed.

When the notification is sent to the registered process, its registration shall be removed. The
message queue shall then be available for registration.

If a process has registered for notification of message arrival at a message queue and some
thread is blocked in mq_receive() waiting to receive a message when a message arrives at the
queue, the arriving message shall satisfy the appropriate mq_receive(). The resulting behavior is
as if the message queue remains empty, and no notification shall be sent.

RETURN VALUE
Upon successful completion, the mq_notify() function shall return a value of zero; otherwise, the
function shall return a value of −1 and set errno to indicate the error.

ERRORS
The mq_notify() function shall fail if:

[EBADF] The mqdes argument is not a valid message queue descriptor.

[EBUSY] A process is already registered for notification by the message queue.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
mq_open(), mq_send(), mq_timedsend(), msgctl(), msgget(), msgrcv(), msgsnd(), the Base
Definitions volume of IEEE Std 1003.1-2001, <mqueue.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.
The `mq_notify()` function is marked as part of the Message Passing option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The `mq_timedsend()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
NAME
mq_open — open a message queue (REALTIME)

SYNOPSIS
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...);

DESCRIPTION
The `mq_open()` function shall establish the connection between a process and a message queue with a message queue descriptor. It shall create an open message queue description that refers to the message queue, and a message queue descriptor that refers to that open message queue description. The message queue descriptor is used by other functions to refer to that message queue. The `name` argument points to a string naming a message queue. 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 `mq_open()` with the same value of `name` shall refer to the same message queue 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. If the `name` argument is not the name of an existing message queue and creation is not requested, `mq_open()` shall fail and return an error.

A message queue descriptor may be implemented using a file descriptor, in which case applications can open up to at least `{OPEN_MAX}` file and message queues.

The `oflag` argument requests the desired receive and/or send access to the message queue. The requested access permission to receive messages or send messages shall be granted if the calling process would be granted read or write access, respectively, to an equivalently protected file.

The value of `oflag` is the bitwise-inclusive OR of values from the following list. Applications shall specify exactly one of the first three values (access modes) below in the value of `oflag`:

- O_RDONLY: Open the message queue for receiving messages. The process can use the returned message queue descriptor with `mq_receive()`, but not `mq_send()`. A message queue may be open multiple times in the same or different processes for receiving messages.

- O_WRONLY: Open the queue for sending messages. The process can use the returned message queue descriptor with `mq_send()` but not `mq_receive()`. A message queue may be open multiple times in the same or different processes for sending messages.

- O_RDWR: Open the queue for both receiving and sending messages. The process can use any of the functions allowed for O_RDONLY and O_WRONLY. A message queue may be open multiple times in the same or different processes for sending messages.

Any combination of the remaining flags may be specified in the value of `oflag`:

- O_CREAT: Create a message queue. It requires two additional arguments: `mode`, which shall be of type `mode_t`, and `attr`, which shall be a pointer to an `mq_attr` structure. If the pathname `name` has already been used to create a message queue that still exists, then this flag shall have no effect, except as noted under O_EXCL. Otherwise, a message queue shall be created without any messages in it. The user ID of the message queue shall be set to the effective user ID of the process, and the group ID of the message queue shall be set to the effective
mq_open() group ID of the process. The file permission bits shall be set to the value of
mode. When bits in mode other than file permission bits are set, the effect is
implementation-defined. If attr is NULL, the message queue shall be created
with implementation-defined default message queue attributes. If attr is non-
NULL and the calling process has the appropriate privilege on name, the
message queue mq_maxmsg and mq_msgsize attributes shall be set to the values
of the corresponding members in the mq_attr structure referred to by attr. If
attr is non-NULL, but the calling process does not have the appropriate
privilege on name, the mq_open() function shall fail and return an error
without creating the message queue.

O_EXCL If O_EXCL and O_CREAT are set, mq_open() shall fail if the message queue
name 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 mq_open() naming the same name with O_EXCL and
O_CREAT set. If O_EXCL is set and O_CREAT is not set, the result is
undefined.

O_NONBLOCK Determines whether an mq_send() or mq_receive() waits for resources or
messages that are not currently available, or fails with errno set to [EAGAIN];
see mq_send() and mq_receive() for details.

The mq_open() function does not add or remove messages from the queue.

RETURN VALUE
Upon successful completion, the function shall return a message queue descriptor; otherwise,
the function shall return (mqd_t)−1 and set errno to indicate the error.

ERRORS
The mq_open() function shall fail if:

[EACCES] The message queue exists and the permissions specified by oflag are denied, or
the message queue does not exist and permission to create the message queue
is denied.

[EEXIST] O_CREAT and O_EXCL are set and the named message queue already exists.

[EINTR] The mq_open() function was interrupted by a signal.

[EINVAL] The mq_open() function is not supported for the given name.

[EINVAL] O_CREAT was specified in oflag, the value of attr is not NULL, and either
mq_maxmsg or mq_msgsize was less than or equal to zero.

[EMFILE] Too many message queue descriptors or file descriptors are currently in use by
this process.

[ENAMETOOLONG] The length of the name argument exceeds [PATH_MAX] or a pathname
component is longer than [NAME_MAX].

[ENFILE] Too many message queues are currently open in the system.

[ENOENT] O_CREAT is not set and the named message queue does not exist.

[ENOSPC] There is insufficient space for the creation of the new message queue.
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
mq_close(), mq_getattr(), mq_receive(), mq_send(), mq_setattr(), mq_timedreceive(), mq_timedsend(), mq_unlink(), msgctl(), msgget(), msgrcv(), msgsnd(), the Base Definitions volume of IEEE Std 1003.1-2001, <mqueue.h>

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

Issue 6
The mq_open() function is marked as part of the Message Passing option.
The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.
The mq_timedreceive() and mq_timedsend() functions are added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48.
NAME

mq_receive, mq_timedreceive — receive a message from a message queue (REALTIME)

SYNOPSIS

```c
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio);
```

```c
#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr, size_t msg_len, unsigned *restrict msg_prio,
                const struct timespec *restrict abs_timeout);
```

DESCRIPTION

The `mq_receive()` function shall receive the oldest of the highest priority message(s) from the message queue specified by `mqdes`. If the size of the buffer in bytes, specified by the `msg_len` argument, is less than the `mq_msgsize` attribute of the message queue, the function shall fail and return an error. Otherwise, the selected message shall be removed from the queue and copied to the buffer pointed to by the `msg_ptr` argument.

If the value of `msg_len` is greater than `{SSIZE_MAX}`, the result is implementation-defined.

If the argument `msg_prio` is not NULL, the priority of the selected message shall be stored in the location referenced by `msg_prio`.

If the specified message queue is empty and O_NONBLOCK is not set in the message queue description associated with `mqdes`, `mq_receive()` shall block until a message is enqueued on the message queue or until `mq_receive()` is interrupted by a signal. If more than one thread is waiting to receive a message when a message arrives at an empty queue and the Priority Scheduling option is supported, then the thread of highest priority that has been waiting the longest shall be selected to receive the message. Otherwise, it is unspecified which waiting thread receives the message. If the specified message queue is empty and O_NONBLOCK is set in the message queue description associated with `mqdes`, no message shall be removed from the queue, and `mq_receive()` shall return an error.

The `mq_timedreceive()` function shall receive the oldest of the highest priority messages from the message queue specified by `mqdes` as described for the `mq_receive()` function. However, if O_NONBLOCK was not specified when the message queue was opened via the `mq_open()` function, and no message exists on the queue to satisfy the receive, the wait for such a message shall be terminated when the specified timeout expires. If O_NONBLOCK is set, this function is equivalent to `mq_receive()`.

The timeout expires 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.

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` argument is defined in the `<time.h>` header.
Under no circumstance shall the operation fail with a timeout if a message can be removed from
the message queue immediately. The validity of the abs_timeout parameter need not be checked
if a message can be removed from the message queue immediately.

RETURN VALUE
Upon successful completion, the mq_receive() and mq_timedreceive() functions shall return the
length of the selected message in bytes and the message shall be removed from the queue.
Otherwise, no message shall be removed from the queue, the functions shall return a value of −1,
and set errno to indicate the error.

ERRORS
The mq_receive() and mq_timedreceive() functions shall fail if:

[EAGAIN] O_NONBLOCK was set in the message description associated with mqdes,
and the specified message queue is empty.
[EBADF] The mqdes argument is not a valid message queue descriptor open for reading.
[EMSGSIZE] The specified message buffer size, msg_len, is less than the message size
attribute of the message queue.
[EINTR] The mq_receive() or mq_timedreceive() operation was interrupted by a signal.
[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.
[ETIMEDOUT] The O_NONBLOCK flag was not set when the message queue was opened,
but no message arrived on the queue before the specified timeout expired.

The mq_receive() and mq_timedreceive() functions may fail if:

[EBADMSG] The implementation has detected a data corruption problem with the
message.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
mq_open(), mq_send(), mq_timedsend(), msgctl(), msgget(), msgrcv(), msgsnd(), time(), the Base
Definitions volume of IEEE Std 1003.1-2001, <mqueue.h>, <time.h>

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

Issue 6
The mq_receive() function is marked as part of the Message Passing option.
The Open Group Corrigendum U021/4 is applied. The DESCRIPTION is changed to refer to
msg_len rather than maxsize.
The [ENOSYS] error condition has been removed as stubs need not be provided if an
implementation does not support the Message Passing option.
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In this function it is possible for the return value to exceed the range of the type ssize_t (since size_t has a larger range of positive values than ssize_t). A sentence restricting the size of the size_t object is added to the description to resolve this conflict.

The mq_timedreceive() function is added for alignment with IEEE Std 1003.1d-1999.

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

IEEE PASC Interpretation 1003.1 #109 is applied, correcting the return type for mq_timedreceive() from int to ssize_t.
NAME
mq_send, mq_timedsend — send a message to a message queue (REALTIME)

SYNOPSIS

#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
unsigned msg_prio);

#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
unsigned msg_prio, const struct timespec *abs_timeout);

DESCRIPTION

The mq_send() function shall add the message pointed to by the argument msg_ptr to the message queue specified by mqdes. The msg_len argument specifies the length of the message, in bytes, pointed to by msg_ptr. The value of msg_len shall be less than or equal to the mq_msgsize attribute of the message queue, or mq_send() shall fail.

If the specified message queue is not full, mq_send() shall behave as if the message is inserted into the message queue at the position indicated by the msg_prio argument. A message with a larger numeric value of msg_prio shall be inserted before messages with lower values of msg_prio. A message shall be inserted after other messages in the queue, if any, with equal msg_prio. The value of msg_prio shall be less than {MQ_PRIO_MAX}.

If the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with mqdes, mq_send() shall block until space becomes available to enqueue the message, or until mq_send() is interrupted by a signal. If more than one thread is waiting to send when space becomes available in the message queue and the Priority Scheduling option is supported, then the thread of the highest priority that has been waiting the longest shall be unblocked to send its message. Otherwise, it is unspecified which waiting thread is unblocked. If the specified message queue is full and O_NONBLOCK is set in the message queue description associated with mqdes, the message shall not be queued and mq_send() shall return an error.

The mq_timedsend() function shall add a message to the message queue specified by mqdes in the manner defined for the mq_send() function. However, if the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with mqdes, the wait for sufficient room in the queue shall be terminated when the specified timeout expires. If O_NONBLOCK is set in the message queue description, this function shall be equivalent to mq_send().

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.

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 argument is defined in the <time.h> header.
mq_send()

Under no circumstance shall the operation fail with a timeout if there is sufficient room in the queue to add the message immediately. The validity of the abs_timeout parameter need not be checked when there is sufficient room in the queue.

RETURN VALUE

Upon successful completion, the mq_send() and mq_timedsend() functions shall return a value of zero. Otherwise, no message shall be enqueued, the functions shall return −1, and errno shall be set to indicate the error.

ERRORS

The mq_send() and mq_timedsend() functions shall fail if:

[EAGAIN] The O_NONBLOCK flag is set in the message queue description associated with mqdes, and the specified message queue is full.

[EBADF] The mqdes argument is not a valid message queue descriptor open for writing.

[EINTR] A signal interrupted the call to mq_send() or mq_timedsend().

EINVAL] The value of msg_prio was outside the valid range.

[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.

EMSGSIZE] The specified message length, msg_len, exceeds the message size attribute of the message queue.

ETIMEDOUT] The O_NONBLOCK flag was not set when the message queue was opened, but the timeout expired before the message could be added to the queue.

EXAMPLES

None.

APPLICATION USAGE

The value of the symbol {MQ_PRIO_MAX} limits the number of priority levels supported by the application. Message priorities range from 0 to {MQ_PRIO_MAX}−1.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_open(), mq_receive(), mq_setattr(), mq_timedreceive(), time(), the Base Definitions volume of IEEE Std 1003.1-2001, <mq.h>, <time.h>

CHANGE HISTORY

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

Issue 6

The mq_send() function is marked as part of the Message Passing option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The mq_timedsend() function is added for alignment with IEEE Std 1003.1d-1999.
NAME
mq_setattr — set message queue attributes (REALTIME)

SYNOPSIS
#include <mqueue.h>

int mq_setattr(mqd_t mqdes, const struct mq_attr *restrict mqstat,
               struct mq_attr *restrict omqstat);

DESCRIPTION
The mq_setattr() function shall set attributes associated with the open message queue
description referenced by the message queue descriptor specified by mqdes.

The message queue attributes corresponding to the following members defined in the mq_attr
structure shall be set to the specified values upon successful completion of mq_setattr():

mq_flags The value of this member is the bitwise-logical OR of zero or more of
O_NONBLOCK and any implementation-defined flags.

The values of the mq_maxmsg, mq_msgsize, and mq_curmsgs members of the mq_attr structure
shall be ignored by mq_setattr().

If omqstat is non-NULL, the mq_setattr() function shall store, in the location referenced by
omqstat, the previous message queue attributes and the current queue status. These values shall
be the same as would be returned by a call to mq_getattr() at that point.

RETURN VALUE
Upon successful completion, the function shall return a value of zero and the attributes of the
message queue shall have been changed as specified.

Otherwise, the message queue attributes shall be unchanged, and the function shall return a
value of −1 and set errno to indicate the error.

ERRORS
The mq_setattr() function shall fail if:

[EBADF] The mqdes argument is not a valid message queue descriptor.

EXCEPTIONS
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
mq_open(), mq_send(), mq_timedsend(), msgctl(), msgget(), msgrcv(), msgsnd(), the Base
Definitions volume of IEEE Std 1003.1-2001, <mqueue.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.
The `mq_setattr()` function is marked as part of the Message Passing option.

The `[ENOSYS]` error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The `mq_timedsend()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

The `restrict` keyword is added to the `mq_setattr()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
mq_timedreceive — receive a message from a message queue (ADVANCED REALTIME)

SYNOPSIS
#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
  size_t msg_len, unsigned *restrict msg_prio,
  const struct timespec *restrict abs_timeout);

DESCRIPTION
Refer to mq_receive().
**NAME**

mq_timedsend — send a message to a message queue (ADVANCED REALTIME)

**SYNOPSIS**

```c
#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
                 unsigned msg_prio, const struct timespec *abs_timeout);
```

**DESCRIPTION**

Refer to `mq_send()`.
NAME
mq_unlink — remove a message queue (REALTIME)

SYNOPSIS
#include <mqueue.h>

int mq_unlink(const char *name);

DESCRIPTION
The mq_unlink() function shall remove the message queue named by the pathname name. After
a successful call to mq_unlink() with name, a call to mq_open() with name shall fail if the flag
O_CREAT is not set in flags. If one or more processes have the message queue open when
mq_unlink() is called, destruction of the message queue shall be postponed until all references to
the message queue have been closed.
Calls to mq_open() to recreate the message queue may fail until the message queue is actually
removed. However, the mq_unlink() call need not block until all references have been closed; it
may return immediately.

RETURN VALUE
Upon successful completion, the function shall return a value of zero. Otherwise, the named
message queue shall be unchanged by this function call, and the function shall return a value of
−1 and set errno to indicate the error.

ERRORS
The mq_unlink() function shall fail if:
[EACCES] Permission is denied to unlink the named message queue.
[ENAMETOOLONG] The length of the name argument exceeds [PATH_MAX] or a pathname
component is longer than [NAME_MAX].
[ENOENT] The named message queue does not exist.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
mq_close(), mq_open(), msgctl(), msgget(), msgrcv(), msgsnd(), the Base Definitions volume of
IEEE Std 1003.1-2001, <mqueue.h>

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

Issue 6
The mq_unlink() function is marked as part of the Message Passing option.
The Open Group Corrigendum U021/5 is applied, clarifying that upon unsuccessful completion,
the named message queue is unchanged by this function.
The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.
NAME

mrand48 — generate uniformly distributed pseudo-random signed long integers

SYNOPSIS

XSI

```
#include <stdlib.h>

long mrand48(void);
```

DESCRIPTION

Refer to `drand48()`.
NAME

msgctl — XSI message control operations

SYNOPSIS

#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);

DESCRIPTION

The msgctl() function operates on XSI message queues (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.224, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 41).

The msgctl() function shall provide message control operations as specified by cmd. The following values for cmd, and the message control operations they specify, are:

IPC_STAT Place the current value of each member of the msqid_ds data structure associated with msqid into the structure pointed to by buf. The contents of this structure are defined in <sys/msg.h>.

IPC_SET Set the value of the following members of the msqid_ds data structure associated with msqid to the corresponding value found in the structure pointed to by buf:

msg_perm.uid
msg_perm.gid
msg_perm.mode
msg_qbytes

IPC_SET can only be executed by a process with appropriate privileges or that has an effective user ID equal to the value of msg_perm.cuid or msg_perm.uid in the msqid_ds data structure associated with msqid. Only a process with appropriate privileges can raise the value of msg_qbytes.

IPC_RMID Remove the message queue identifier specified by msqid from the system and destroy the message queue and msqid_ds data structure associated with it. IPC_RMID can only be executed by a process with appropriate privileges or one that has an effective user ID equal to the value of msg_perm.cuid or msg_perm.uid in the msqid_ds data structure associated with msqid.

RETURN VALUE

Upon successful completion, msgctl() shall return 0; otherwise, it shall return −1 and set errno to indicate the error.

ERRORS

The msgctl() function shall fail if:

[EACCES] The argument cmd is IPC_STAT and the calling process does not have read permission; see Section 2.7 (on page 39).

[EINVAL] The value of msqid is not a valid message queue identifier; or the value of cmd is not a valid command.

[EPERM] The argument cmd is 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 msg_perm.cuid or msg_perm.uid in the data structure associated with msqid.
The argument cmd is IPC_SET, an attempt is being made to increase to the value of msg_qbytes, and the effective user ID of the calling process does not have appropriate privileges.

**EXAMPLES**

None.

**APPLICATION USAGE**

The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). 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.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

Section 2.7 (on page 39), Section 2.8 (on page 41), mq_close(), mq_getattr(), mq_notify(), mq_open(), mq_receive(), mq_send(), mq_setattr(), mq_unlink(), msgget(), msggetv(), msgrcv(), msqsend(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/msg.h>

**CHANGE HISTORY**

First released in Issue 2. Derived from Issue 2 of the SVID.

**Issue 5**

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.
NAME
msgget — get the XSI message queue identifier

SYNOPSIS
XSI
#include <sys/msg.h>

int msgget(key_t key, int msgflg);

DESCRIPTION
The msgget() function operates on XSI message queues (see the Base Definitions volume of
IEEE Std 1003.1-2001, Section 3.224, Message Queue). It is unspecified whether this function
interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
page 41).

The msgget() function shall return the message queue identifier associated with the argument
key.

A message queue identifier, associated message queue, and data structure (see <sys/msg.h>),
shall be created for the argument key if one of the following is true:

• The argument key is equal to IPC_PRIVATE.
• The argument key does not already have a message queue identifier associated with it, and
  (msgflg & IPC_CREAT) is non-zero.

Upon creation, the data structure associated with the new message queue identifier shall be
initialized as follows:

• msg_perm.cuid, msg_perm.uid, msg_perm.cgid, and msg_perm.gid shall be set equal to the
effective user ID and effective group ID, respectively, of the calling process.
• The low-order 9 bits of msg_perm.mode shall be set equal to the low-order 9 bits of msgflg.
• msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime shall be set equal to 0.
• msg_c_time shall be set equal to the current time.
• msg_qbytes shall be set equal to the system limit.

RETURN VALUE
Upon successful completion, msgget() shall return a non-negative integer, namely a message
queue identifier. Otherwise, it shall return −1 and set errno to indicate the error.

ERRORS
The msgget() function shall fail if:

[EACCES] A message queue identifier exists for the argument key, but operation
permission as specified by the low-order 9 bits of msgflg would not be granted;
see Section 2.7 (on page 39).

[EXIST] A message queue identifier exists for the argument key but ((msgflg &
IPC_CREAT) & (msgflg & IPC_EXCL)) is non-zero.

[ENOENT] A message queue identifier does not exist for the argument key and (msgflg &
IPC_CREAT) is 0.

[ENOSPC] A message queue identifier is to be created but the system-imposed limit on
the maximum number of allowed message queue identifiers system-wide
would be exceeded.
EXAMPLES
None.

APPLICATION USAGE
The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). 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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.7 (on page 39), Section 2.8 (on page 41), mq_close(), mq_getattr(), mq_notify(), mq_open(), mq_receive(), mq_send(), mq_setattr(), mq_unlink(), msgctl(), msgrcv(), msgsnd(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/msg.h>

CHANGE HISTORY
First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5
The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.
msgrcv()

NAME
msgrcv — XSI message receive operation

SYNOPSIS
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
    int msgflg);

DESCRIPTION
The msgrcv() function operates on XSI message queues (see the Base Definitions volume of
IEEE Std 1003.1-2001, Section 3.224, Message Queue). It is unspecified whether this function
interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
page 41).

The msgrcv() function shall read a message from the queue associated with the message queue
identifier specified by msqid and place it in the user-defined buffer pointed to by msgp.

The application shall ensure that the argument msgp points to a user-defined buffer that contains
first a field of type long specifying the type of the message, and then a data portion that holds
the data bytes of the message. The structure below is an example of what this user-defined
buffer might look like:

struct mymsg {
    long mtype; /* Message type. */
    char mtext[1]; /* Message text. */
}

The structure member mtype is the received message's type as specified by the sending process.
The structure member mtext is the text of the message.

The argument msgsz specifies the size in bytes of mtext. The received message shall be truncated
to msgsz bytes if it is larger than msgsz and (msgflg & MSG_NOERROR) is non-zero. The
truncated part of the message shall be lost and no indication of the truncation shall be given to
the calling process.

If the value of msgsz is greater than {SSIZE_MAX}, the result is implementation-defined.

The argument msgtyp specifies the type of message requested as follows:

- If msgtyp is 0, the first message on the queue shall be received.
- If msgtyp is greater than 0, the first message of type msgtyp shall be received.
- If msgtyp is less than 0, the first message of the lowest type that is less than or equal to the
  absolute value of msgtyp shall be received.

The argument msgflg specifies the action to be taken if a message of the desired type is not on the
queue. These are as follows:

- If (msgflg & IPC_NOWAIT) is non-zero, the calling thread shall return immediately with a
  return value of −1 and errno set to [ENOMSG].
- If (msgflg & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the
  following occurs:
    - A message of the desired type is placed on the queue.
    - The message queue identifier msqid is removed from the system; when this occurs, errno
      shall be set equal to [EIDRM] and −1 shall be returned.
The calling thread receives a signal that is to be caught; in this case a message is not received and the calling thread resumes execution in the manner prescribed in \textit{sigaction}().

Upon successful completion, the following actions are taken with respect to the data structure associated with \textit{msqid}:

- \textit{msg\_qnum} shall be decremented by 1.
- \textit{msg\_lrpid} shall be set equal to the process ID of the calling process.
- \textit{msg\_rtime} shall be set equal to the current time.

**RETURN VALUE**

Upon successful completion, \textit{msgrcv()} shall return a value equal to the number of bytes actually placed into the buffer \textit{mtext}. Otherwise, no message shall be received, \textit{msgrcv()} shall return \texttt{(-1)} and \textit{errno} shall be set to indicate the error.

**ERRORS**

The \textit{msgrcv()} function shall fail if:

- [E2BIG] The value of \textit{mtext} is greater than \textit{msgsz} and \((\textit{msgflg} \& \text{MSG\_NOERROR})\) is 0.
- [EACCES] Operation permission is denied to the calling process; see Section 2.7 (on page 39).
- [EIDRM] The message queue identifier \textit{msqid} is removed from the system.
- [EINTR] The \textit{msgrcv()} function was interrupted by a signal.
- [EINVAL] \textit{msqid} is not a valid message queue identifier.
- [ENOMEM] The queue does not contain a message of the desired type and \((\textit{msgflg} \& \text{IPC\_NOWAIT})\) is non-zero.

**EXAMPLES**

### Receiving a Message

The following example receives the first message on the queue (based on the value of the \textit{msgtyp} argument, 0). The queue is identified by the \textit{msqid} argument (assuming that the value has previously been set). This call specifies that an error should be reported if no message is available, but not if the message is too large. The message size is calculated directly using the \texttt{sizeof} operator.

```c
#include <sys/msg.h>
...
int result;
int msqid;
struct message {
    long type;
    char text[20];
} msg;
long msgtyp = 0;
...
result = msgrcv(msqid, (void *) &msg, sizeof(msg.text),
               msgtyp, MSG\_NOERROR | IPC\_NOWAIT);
```
msgrcv()

APPLICATION USAGE
The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). 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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.7 (on page 39), Section 2.8 (on page 41), mq_close(), mq_getattr(), mq_notify(), mq_open(), mq_receive(), mq_send(), mq_setattr(), mq_unlink(), msgctl(), msgget(), msgsnd(), sigaction(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/msg.h>

CHANGE HISTORY
First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5
The type of the return value is changed from int to ssize_t, and a warning is added to the DESCRIPTION about values of msgsz larger than {SSIZE_MAX}.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to the APPLICATION USAGE section.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
msgsnd — XSI message send operation

SYNOPSIS
XSI
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

DESCRIPTION
The msgsnd() function operates on XSI message queues (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.224, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 41).

The msgsnd() function shall send a message to the queue associated with the message queue identifier specified by msqid.

The application shall ensure that the argument msgp points to a user-defined buffer that contains first a field of type long specifying the type of the message, and then a data portion that holds the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

struct mymsg {
    long mtype; /* Message type. */
    char mtext[1]; /* Message text. */
}

The structure member mtype is a non-zero positive type long that can be used by the receiving process for message selection.

The structure member mtext is any text of length msgsz bytes. The argument msgsz can range from 0 to a system-imposed maximum.

The argument msgflg specifies the action to be taken if one or more of the following is true:

- The number of bytes already on the queue is equal to msg_qbytes; see <sys/msg.h>.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (msgflg & IPC_NOWAIT) is non-zero, the message shall not be sent and the calling thread shall return immediately.

- If (msgflg & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The message queue identifier msqid is removed from the system; when this occurs, errno shall be set equal to [EIDRM] and −1 shall be returned.
  - The calling thread receives a signal that is to be caught; in this case the message is not sent and the calling thread resumes execution in the manner prescribed in sigaction().

Upon successful completion, the following actions are taken with respect to the data structure associated with msqid; see <sys/msg.h>.
• **msg_qnum** shall be incremented by 1.
• **msg_lspid** shall be set equal to the process ID of the calling process.
• **msg_stime** shall be set equal to the current time.

**RETURN VALUE**
Upon successful completion, `msgsnd()` shall return 0; otherwise, no message shall be sent, `msgsnd()` shall return −1, and `errno` shall be set to indicate the error.

**ERRORS**
The `msgsnd()` function shall fail if:

- **[EACCES]** Operation permission is denied to the calling process; see Section 2.7 (on page 39).
- **[EAGAIN]** The message cannot be sent for one of the reasons cited above and (`msgflg & IPC_NOWAIT`) is non-zero.
- **[EIDRM]** The message queue identifier `msgid` is removed from the system.
- **[EINVAL]** The value of `msqid` is not a valid message queue identifier, or the value of `mtype` is less than 1; or the value of `msgsz` is less than 0 or greater than the system-imposed limit.

**EXAMPLES**

**Sending a Message**
The following example sends a message to the queue identified by the `msqid` argument (assuming that value has previously been set). This call specifies that an error should be reported if no message is available. The message size is calculated directly using the `sizeof` operator.

```c
#include <sys/msg.h>
...
int result;
int msqid;
struct message {
    long type;
    char text[20];
} msg;
msg.type = 1;
strcpy(msg.text, "This is message 1");
... 
result = msgsnd(msqid, (void *) &msg, sizeof(msg.text), IPC_NOWAIT);
```

**APPLICATION USAGE**
The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). 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.
System Interfaces

msgsnd()

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.7 (on page 39), Section 2.8 (on page 41), mq_close(), mq_getattr(), mq_notify(), mq_open(), mq_receive(), mq_send(), mq Attr(), mq_unlink(), msgctl(), msgget(), msgrcv(), sigaction(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/msg.h>

CHANGE HISTORY
First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5
The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
msync — synchronize memory with physical storage

SYNOPSIS
#include <sys/mman.h>

int msync(void *addr, size_t len, int flags);

DESCRIPTION
The msync() function shall write all modified data to permanent storage locations, if any, in
those whole pages containing any part of the address space of the process starting at address
addr and continuing for len bytes. If no such storage exists, msync() need not have any effect. If
requested, the msync() function shall then invalidate cached copies of data.

The implementation shall require that addr be a multiple of the page size as returned by
sysconf().

For mappings to files, the msync() function shall ensure that all write operations are completed
as defined for synchronized I/O data integrity completion. It is unspecified whether the
implementation also writes out other file attributes. When the msync() function is called on
MAP_PRIVATE mappings, any modified data shall not be written to the underlying object and
shall not cause such data to be made visible to other processes. It is unspecified whether data in
MAP_PRIVATE mappings has any permanent storage locations. The effect of msync() on a
shared memory object or a typed memory object is unspecified. The behavior of this function is
unspecified if the mapping was not established by a call to mmap().

The flags argument is constructed from the bitwise-inclusive OR of one or more of the following
flags defined in the <sys/mman.h> header:

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSASYNC</td>
<td>Perform asynchronous writes.</td>
</tr>
<tr>
<td>MS_SYNC</td>
<td>Perform synchronous writes.</td>
</tr>
<tr>
<td>MSINVALIDATE</td>
<td>Invalidate cached data.</td>
</tr>
</tbody>
</table>

When MSASYNC is specified, msync() shall return immediately once all the write operations
are initiated or queued for servicing; when MS_SYNC is specified, msync() shall not return until
all write operations are completed as defined for synchronized I/O data integrity completion.
Either MSASYNC or MS_SYNC is specified, but not both.

When MSINVALIDATE is specified, msync() shall invalidate all cached copies of mapped data
that are inconsistent with the permanent storage locations such that subsequent references shall
obtain data that was consistent with the permanent storage locations sometime between the call
to msync() and the first subsequent memory reference to the data.

If msync() causes any write to a file, the file’s st_ctime and st_mtime fields shall be marked for
update.

RETURN VALUE
Upon successful completion, msync() shall return 0; otherwise, it shall return −1 and set errno to
indicate the error.

ERRORS
The msync() function shall fail if:

[EBUSY] Some or all of the addresses in the range starting at addr and continuing for len
bytes are locked, and MSINVALIDATE is specified.
The value of *flags* is invalid.

The value of *addr* is not a multiple of the page size \(\text{PAGESIZE}\).

The addresses in the range starting at *addr* and continuing for *len* bytes are outside the range allowed for the address space of a process or specify one or more pages that are not mapped.

**EXAMPLES**

None.

**APPLICATION USAGE**

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.

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.

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.

**RATIONALE**

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.

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.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`mmap()`, `sysconf()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/mman.h>`

**CHANGE HISTORY**

First released in Issue 4, Version 2.

**Issue 5**

Moved from X/OPEN UNIX extension to BASE.

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.

**Issue 6**

The `msync()` function is marked as part of the Memory Mapped Files and Synchronized Input and Output options.

The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- The [EBUSY] mandatory error condition is added.

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.
msync()

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding reference to typed memory objects.
munlock — unlock a range of process address space

SYNOPSIS

```c
#include <sys/mman.h>

int munlock(const void *addr, size_t len);
```

DESCRIPTION

Refer to `mlock()`.
munlockall() — unlock the address space of a process

**NAME**
munlockall — unlock the address space of a process

**SYNOPSIS**
```c
#include <sys/mman.h>

int munlockall(void);
```

**DESCRIPTION**
Refer to `mlockall()`.
**NAME**
munmap — unmap pages of memory

**SYNOPSIS**

```c
#include <sys/mman.h>

int munmap(void *addr, size_t len);
```

**DESCRIPTION**

The `munmap()` function shall remove any mappings for those entire pages containing any part of the address space of the process starting at `addr` and continuing for `len` bytes. Further references to these pages shall result in the generation of a SIGSEGV signal to the process. If there are no mappings in the specified address range, then `munmap()` has no effect.

The implementation shall require that `addr` be a multiple of the page size `{PAGESIZE}`.

If a mapping to be removed was private, any modifications made in this address range shall be discarded.

Any memory locks (see `mlock()` and `mlockall()`) associated with this address range shall be removed, as if by an appropriate call to `munlock()`.

If a mapping removed from a typed memory object causes the corresponding address range of the memory pool to be inaccessible by any process in the system except through allocatable mappings (that is, mappings of typed memory objects opened with the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag), then that range of the memory pool shall become deallocated and may become available to satisfy future typed memory allocation requests.

A mapping removed from a typed memory object opened with the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag shall not affect in any way the availability of that typed memory for allocation.

The behavior of this function is unspecified if the mapping was not established by a call to `mmap()`.

**RETURN VALUE**

Upon successful completion, `munmap()` shall return 0; otherwise, it shall return −1 and set `errno` to indicate the error.

**ERRORS**

The `munmap()` function shall fail if:

- [EINVAL] Addresses in the range `[addr, addr+len)` are outside the valid range for the address space of a process.
- [EINVAL] The `len` argument is 0.
- [EINVAL] The `addr` argument is not a multiple of the page size as returned by `sysconf()`.
munmap()

EXAMPLES
None.

APPLICATION USAGE
The munmap() function is only supported if the Memory Mapped Files option or the Shared Memory Objects option is supported.

RATIONALE
The munmap() function corresponds to SVR4, just as the mmap() function does.

APPLICATION USAGE
It is possible that an application has applied process memory locking to a region that contains shared memory. If this has occurred, the munmap() call ignores those locks and, if necessary, causes those locks to be removed.

FUTURE DIRECTIONS
None.

SEE ALSO
mlock(), mlockall(), mmap(), posix_typed_mem_open(), sysconf(), the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>, <sys/mman.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Moved from X/OPEN UNIX extension to BASE.

Aligned with munmap() in the POSIX Realtime Extension as follows:

• The DESCRIPTION is extensively reworded.

• The SIGBUS error is no longer permitted to be generated.

The munmap() function is marked as part of the Memory Mapped Files and Shared Memory Objects option.

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 [EINVAL] error conditions are added.

The following changes are made for alignment with IEEE Std 1003.1j-2000:

• Semantics for typed memory objects are added to the DESCRIPTION.

• The posix_typed_mem_open() function is added to the SEE ALSO section.
NAME
nan, nanf, nanl — return quiet NaN

SYNOPSIS
#include <math.h>

double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);

DESCRIPTION
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 function call nan("n-char-sequence") shall be equivalent to:

```c
strtod("NAN(n-char-sequence)", (char **) NULL);
```

The function call nan("") shall be equivalent to:

```c
strtod("NAN()", (char **) NULL)
```

If tagp does not point to an n-char sequence or an empty string, the function call shall be equivalent to:

```c
strtod("NAN", (char **) NULL)
```

Function calls to nanf() and nanl() are equivalent to the corresponding function calls to strtof() and strtold().

RETURN VALUE
These functions shall return a quiet NaN, if available, with content indicated through tagp.

If the implementation does not support quiet NaNs, these functions shall return zero.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strtod(), strtold(), the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

CHANGE HISTORY
nanosleep( )

NAME
nanosleep — high resolution sleep (REALTIME)

SYNOPSIS
#include <time.h>

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);

DESCRIPTION
The nanosleep() function shall cause the current thread to be suspended from execution until either the time interval specified by the rqtp argument has elapsed or a signal is delivered to the calling thread, and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time shall not be less than the time specified by rqtp, as measured by the system clock CLOCK_REALTIME.

The use of the nanosleep() function has no effect on the action or blockage of any signal.

RETURN VALUE
If the nanosleep() function returns because the requested time has elapsed, its return value shall be zero.

If the nanosleep() function returns because it has been interrupted by a signal, it shall return a value of −1 and set errno to indicate the interruption. If the rmtp argument is non-NULL, the timespec structure referenced by it is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the rmtp argument is NULL, the remaining time is not returned.

If nanosleep() fails, it shall return a value of −1 and set errno to indicate the error.

ERRORS
The nanosleep() function shall fail if:

[EINTR] The nanosleep() function was interrupted by a signal.

[EINVAL] The rqtp argument specified a nanosecond value less than zero or greater than or equal to 1 000 million.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
It is common to suspend execution of a process for an interval in order to poll the status of a non-interrupting function. A large number of actual needs can be met with a simple extension to sleep() that provides finer resolution.

In the POSIX.1-1990 standard and SVR4, it is possible to implement such a routine, but the frequency of wakeup is limited by the resolution of the alarm() and sleep() functions. In 4.3 BSD, it is possible to write such a routine using no static storage and reserving no system facilities. Although it is possible to write a function with similar functionality to sleep() using the remainder of the timer_*() functions, such a function requires the use of signals and the reservation of some signal number. This volume of IEEE Std 1003.1-2001 requires that nanosleep() be non-intrusive of the signals function.
The `nanosleep()` function shall return a value of 0 on success and –1 on failure or if interrupted. This latter case is different from `sleep()`. This was done because the remaining time is returned via an argument structure pointer, `rmtp`, instead of as the return value.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`sleep()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<time.h>`

**CHANGE HISTORY**

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

**Issue 6**

The `nanosleep()` function is marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.
nearbyint()

NAME
nearbyint, nearbyintf, nearbyintl — floating-point rounding functions

SYNOPSIS
#include <math.h>

double nearbyint(double x);
f float nearbyintf(float x);
long double nearbyintl(long double x);

DESCRIPTION
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

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.

An application wishing to check for error situations should set errno to zero and call

fearlarexcept(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.

RETURN VALUE
Upon successful completion, these functions shall return the rounded integer value.

If x is NaN, a NaN shall be returned.
If x is ±0, ±0 shall be returned.
If x is ±Inf, x shall be returned.

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.

ERRORS
These functions shall fail if:

Range Error The result would cause an overflow.

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.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO

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

CHANGE HISTORY

NAME
nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, nexttowardl — next representable floating-point number

SYNOPSIS
#include <math.h>

double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);

DESCRIPTION
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 nextafter(), nextafterf(), and nextafterl() functions shall compute the next representable floating-point value following x in the direction of y. Thus, if y is less than x, nextafter() shall return the largest representable floating-point number less than x. The nextafter(), nextafterf(), and nextafterl() functions shall return y if x equals y.

The nexttoward(), nexttowardf(), and nexttowardl() functions shall be equivalent to the corresponding nextafter() functions, except that the second parameter shall have type long double and the functions shall return y converted to the type of the function if x equals y.

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.

RETURN VALUE
Upon successful completion, these functions shall return the next representable floating-point value following x in the direction of y.

If x==y, y (of the type x) shall be returned.

If x is finite and the correct function value would overflow, a range error shall occur and ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (with the same sign as x) shall be returned as appropriate for the return type of the function.

If x or y is NaN, a NaN shall be returned.

If x!=y and the correct function value is subnormal, zero, or underflows, a range error shall occur, and either the correct function value (if representable) or 0.0 shall be returned.

ERRORS
These functions shall fail if:

Range Error The correct value overflows.

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.

Range Error The correct value is subnormal or underflows.
If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then \texttt{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.

**EXAMPLES**

None.

**APPLICATION USAGE**

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.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

\texttt{feklearexcept()}, \texttt{fetestexcept()}, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

**CHANGE HISTORY**

First released in Issue 4, Version 2.

**Issue 5**

Moved from X/OPEN UNIX extension to BASE.

**Issue 6**

The \texttt{nextafter()} function is no longer marked as an extension.

The \texttt{nextafterf()}, \texttt{nextafterl()}, \texttt{nexttoward()}, \texttt{nexttowardf()}, and \texttt{nexttowardl()} functions are added for alignment with the ISO/IEC 9899:1999 standard.

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

NAME
nftw — walk a file tree

SYNOPSIS
#include <ftw.h>

int nftw(const char *path, int (*fn)(const char *,
    const struct stat *, int, struct FTW *), int depth, int flags);

DESCRIPTION
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:

FTW_CHDIR If set, nftw() shall change the current working directory to each directory as it reports files in that directory. If clear, nftw() shall not change the current working directory.

FTW_DEPTH If set, nftw() shall report all files in a directory before reporting the directory itself. If clear, nftw() shall report any directory before reporting the files in that directory.

FTW_MOUNT If set, nftw() shall only report files in the same file system as path. If clear, nftw() shall report all files encountered during the walk.

FTW_PHYS If set, nftw() shall perform a physical walk and shall not follow symbolic links.

If FTW_PHYS is clear and FTW_DEPTH is set, nftw() shall follow links instead of reporting them, but shall not report any directory that would be a descendant of itself. If FTW_PHYS is clear and FTW_DEPTH is clear, nftw() shall follow links instead of reporting them, but shall not report the contents of any directory that would be a descendant of itself.

At each file it encounters, nftw() shall call the user-supplied function fn with four arguments:

- The first argument is the pathname of the object.
- The second argument is a pointer to the stat buffer containing information on the object.
- The third argument is an integer giving additional information. Its value is one of the following:
  - FTW_F The object is a file.
  - FTW_D The object is a directory.
  - FTW_DP The object is a directory and subdirectories have been visited. (This condition shall only occur if the FTW_DEPTH flag is included in flags.)
  - FTW_SL The object is a symbolic link. (This condition shall only occur if the FTW_PHYS flag is included in flags.)
  - FTW_SLN The object is a symbolic link that does not name an existing file. (This condition shall only occur if the FTW_PHYS flag is not included in flags.)
  - FTW_DNR The object is a directory that cannot be read. The fn function shall not be called for any of its descendants.
  - FTW_NS The stat() function failed on the object because of lack of appropriate permission. The stat buffer passed to fn is undefined. Failure of stat() for any other reason is considered an error and nftw() shall return −1.
• The fourth argument is a pointer to an FTW structure. The value of base is the offset of the
  object’s filename in the pathname passed as the first argument to fn. The value of level
  indicates depth relative to the root of the walk, where the root level is 0.

The results are unspecified if the application-supplied fn function does not preserve the current
working directory.

The argument depth sets the maximum number of file descriptors that shall be used by nftw()
while traversing the file tree. At most one file descriptor shall be used for each directory level.

The nftw() function need not be reentrant. A function that is not required to be reentrant is not
required to be thread-safe.

RETURN VALUE

The nftw() function shall continue until the first of the following conditions occurs:

• An invocation of fn shall return a non-zero value, in which case nftw() shall return that value.

• The nftw() function detects an error other than [EACCES] (see FTW_DNR and FTW_NS
  above), in which case nftw() shall return −1 and set errno to indicate the error.

• The tree is exhausted, in which case nftw() shall return 0.

ERRORS

The nftw() function shall fail if:

[EACCES] Search permission is denied for any component of path or read permission is
denied for path, or fn returns −1 and does not reset errno.

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

[ENAMETOOLONG] The length of the path argument exceeds [PATH_MAX] or a pathname
  component is longer than [NAME_MAX].

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

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

[EOVERFLOW] A field in the stat structure cannot be represented correctly in the current
  programming environment for one or more files found in the file hierarchy.

The nftw() function may fail if:

[ELOOP] More than [SYMLOOP_MAX] symbolic links were encountered during
  resolution of the path argument.

[EMFILE] [OPEN_MAX] file descriptors are currently open in the calling process.

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
  whose length exceeds [PATH_MAX].

[ENFILE] Too many files are currently open in the system.

In addition, errno may be set if the function pointed to by fn causes errno to be set.
EXAMPLES

The following example walks the /tmp directory and its subdirectories, calling the nftw() function for every directory entry, to a maximum of 5 levels deep.

```c
#include <ftw.h>
...
int nftwfunc(const char *, const struct stat *, int, struct FTW *);
int nftwfunc(const char *filename, const struct stat *statptr,
    int fileflags, struct FTW *pfwt)
{
    return 0;
}
...
char *startpath = "/tmp";
int depth = 5;
int flags = FTW_CHDIR | FTW_DEPTH | FTW_MOUNT;
int ret;
ret = nftw(startpath, nftwfunc, depth, flags);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

lstat(), opendir(), readdir(), stat(), the Base Definitions volume of IEEE Std 1003.1-2001, <ftw.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

In the DESCRIPTION, the definition of the depth argument is clarified.

Issue 6

The Open Group Base Resolution bwg97-003 is applied.

The ERRORS section is updated as follows:

- The wording of the mandatory [ELOOP] error condition is updated.
- A second optional [ELOOP] error condition is added.
- The [EOVERFLOW] mandatory error condition is added.

Text is added to the DESCRIPTION to say that the nftw() function need not be reentrant and that the results are unspecified if the application-supplied fn function does not preserve the current working directory.
NAME
nice — change the nice value of a process

SYNOPSIS
XSI
#include <unistd.h>

int nice(int incr);

DESCRIPTION
The nice() function shall add the value of incr to the nice value of the calling process. A process’
nice value is a non-negative number for which a more positive value shall result in less favorable
scheduling.

A maximum nice value of 2*[NZERO]–1 and a minimum nice value of 0 shall be imposed by the
system. Requests for values above or below these limits shall result in the nice value being set to
the corresponding limit. Only a process with appropriate privileges can lower the nice value.

Calling the nice() function has no effect on the priority of processes or threads with policy
SCHED_FIFO or SCHED_RR. The effect on processes or threads with other scheduling policies
is implementation-defined.

The nice value set with nice() shall be applied to the process. If the process is multi-threaded, the
nice value shall affect all system scope threads in the process.

As −1 is a permissible return value in a successful situation, an application wishing to check for
error situations should set errno to 0, then call nice(), and if it returns −1, check to see whether
errno is non-zero.

RETURN VALUE
Upon successful completion, nice() shall return the new nice value −{NZERO}. Otherwise, −1
shall be returned, the process’ nice value shall not be changed, and errno shall be set to indicate
the error.

ERRORS
The nice() function shall fail if:

[EPERM] The incr argument is negative and the calling process does not have
appropriate privileges.

EXAMPLES
Changing the Nice Value
The following example adds the value of the incr argument, −20, to the nice value of the calling
process.

#include <unistd.h>
...
int incr = −20;
int ret;
ret = nice(incr);

APPLICATION USAGE
None.
Rationale

None.

Future Directions

None.

See Also

getpriority(), setpriority(), the Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <unistd.h>

Change History

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

Issue 5

A statement is added to the description indicating the effects of this function on the different scheduling policies and multi-threaded processes.
NAME
nl_langinfo — language information

SYNOPSIS
#include <langinfo.h>
char *nl_langinfo(nl_item item);

DESCRIPTION
The nl_langinfo() function shall return a pointer to a string containing information relevant to
the particular language or cultural area defined in the program's locale (see <langinfo.h>). The
manifest constant names and values of item are defined in <langinfo.h>. For example:

nl_langinfo(ABDAY_1)
would return a pointer to the string "Dom" if the identified language was Portuguese, and
"Sun" if the identified language was English.

Calls to setlocale() with a category corresponding to the category of item (see <langinfo.h>), or to
the category LC_ALL, may overwrite the array pointed to by the return value.

The nl_langinfo() function need not be reentrant. A function that is not required to be reentrant is
not required to be thread-safe.

RETURN VALUE
In a locale where langinfo data is not defined, nl_langinfo() shall return a pointer to the
conceivably string in the POSIX locale. In all locales, nl_langinfo() shall return a pointer to an
empty string if item contains an invalid setting.

This pointer may point to static data that may be overwritten on the next call.

ERRORS
No errors are defined.

EXAMPLES

Getting Date and Time Formatting Information
The following example returns a pointer to a string containing date and time formatting
information, as defined in the LC_TIME category of the current locale.

#include <time.h>
#include <langinfo.h>
....strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);
... APPLICATION USAGE
The array pointed to by the return value should not be modified by the program, but may be
modified by further calls to nl_langinfo().

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO

setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <langinfo.h>, <nl_types.h>

CHANGE HISTORY

First released in Issue 2.

Issue 5
The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.
A note indicating that this function need not be reentrant is added to the DESCRIPTION.
nrand48( )

NAME
nrand48 — generate uniformly distributed pseudo-random non-negative long integers

SYNOPSIS
#include <stdlib.h>

long nrand48(unsigned short xsubi[3]);

DESCRIPTION
Refer to drand48().
**ntohl()**

**NAME**

ntohl, ntohs — convert values between host and network byte order

**SYNOPSIS**

```c
#include <arpa/inet.h>

uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

**DESCRIPTION**

Refer to `htonl()`.

---

*System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.*
NAME

open — open a file

SYNOPSIS

#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag, ...);

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:

O_RDONLY Open for reading only.
O_WRONLY Open for writing only.
O_RDWR Open for reading and writing. The result is undefined if this flag is applied to
a FIFO.

Any combination of the following may be used:

O_APPEND If set, the file offset shall be set to the end of the file prior to each write.
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.

O_DSYNC Write I/O operations on the file descriptor shall complete as defined by
synchronized I/O data integrity completion.
open()  System Interfaces

O_EXCL If O_CREAT and O_EXCL are set, open() 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 open() naming the same filename in the same directory with O_EXCL and O_CREAT set. If O_EXCL and O_CREAT are set, and path names a symbolic link, open() shall fail and set errno to [EEXIST], regardless of the contents of the symbolic link. If O_EXCL is set and O_CREAT is not set, the result is undefined.

O_NOCTTY If set and path identifies a terminal device, open() shall not cause the terminal device to become the controlling terminal for the process.

O_NONBLOCK When opening a FIFO with O_RDONLY or O_WRONLY set:

- If O_NONBLOCK is set, an open() for reading-only shall return without delay. An open() for writing-only shall return an error if no process currently has the file open for reading.
- If O_NONBLOCK is clear, an open() for reading-only shall block the calling thread until a thread opens the file for writing. An open() 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:

- If O_NONBLOCK is set, the open() 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 open() function shall block the calling thread until the device is ready or available before returning.

Otherwise, the behavior of O_NONBLOCK is unspecified.

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 oflag, all I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion. If both O_SYNC and O_RSYNC are set in flags, all I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

O_SYNC Write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

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.

If O_CREAT is set and the file did not previously exist, upon successful completion, open() shall mark for update the st_atime, st_ctime, and st_mtime fields of the file and the st_ctime and st_mtime fields of the parent directory.

If O_TRUNC is set and the file did previously exist, upon successful completion, open() shall mark for update the st_ctime and st_mtime fields of the file.
If both the O_SYNC and O_DSYNC flags are set, the effect is as if only the O_SYNC flag was set.

If path refers to a STREAMS file, oflag 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.

If path names the master side of a pseudo-terminal device, then it is unspecified whether open() locks the slave side so that it cannot be opened. Conforming applications shall call unlockpt() before opening the slave side.

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.

RETURN VALUE

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 errno set to indicate the error. No files shall be created or modified if the function returns −1.

ERRORS

The open() function shall fail if:

- [EACCES] Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by oflag 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.

- [EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

- [EINVAL] The implementation does not support synchronized I/O for this file.

- [EIO] The path argument names a STREAMS file and a hangup or error occurred during the open().

- [EISDIR] The named file is a directory and oflag includes O_WRONLY or O_RDWR.

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

- [EMFILE] [OPEN_MAX] file descriptors are currently open in the calling process.

- [ENAMETOOLONG] The length of the path argument exceeds [PATH_MAX] or a pathname component is longer than [NAME_MAX].

- [ENFILE] The maximum allowable number of files is currently open in the system.

- [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 path argument points to an empty string.

- [ENOSR] The path argument names a STREAMS-based file and the system is unable to allocate a STREAM.

- [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.
The `open()` function may fail if:

- **[ENOTDIR]** A component of the path prefix is not a directory.
- **[ENXIO]** `O_NONBLOCK` is set, the named file is a FIFO, `O_WRONLY` is set, and no process has the file open for reading.
- **[ENXIO]** The named file is a character special or block special file, and the device associated with this special file does not exist.
- **[ELOOP]** More than `{SYMLOOP_MAX}` symbolic links were encountered during resolution of the `path` argument.
- **[ENOMEM]** The `path` argument names a STREAMS file and the system is unable to allocate resources.
- **[ENOENT]** The file does not exist.
- **[ENOEXEC]** The file is not an executable file.
- **[EACCES]** The process does not have the appropriate permission to perform the operation on the file.
- **[EFAULT]** The `path` argument is outside the address space of the process.
- **[EFAULT]** The `off_t` value is outside the address space of the process.
- **[EINVAL]** The value of the `oflag` argument is not valid.
- **[EINVAL]** The file exists but is not of type `OFF_T`.
- **[EMFILE]** Too many open file descriptors.
- **[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 `oflag` argument.
- **[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`.
- **[ELOOP]** The `path` argument contains a symbolic link that is referenced more than `{SYMLOOP_MAX}` times.
- **[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}`.
- **[ETXTBSY]** The file is a pure procedure (shared text) file that is being executed and `oflag` is `O_WRONLY` or `O_RDWR`.
- **[EAGAIN]** The `path` argument names the slave side of a pseudo-terminal device that is locked.

### EXAMPLES

**Opening a File for Writing by the Owner**

The following example opens the file `/tmp/file`, either by creating it (if it does not already exist), or by truncating its length to 0 (if it does exist). In the former case, if the call creates a new file, the access permission bits in the file mode of the file are set to permit reading and writing by the owner, and to permit reading only by group members and others.

If the call to `open()` is successful, the file is opened for writing.

```c
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);
...
Opening a File Using an Existence Check

The following example uses the `open()` function to try to create the LOCKFILE file and open it for writing. Since the `open()` function specifies the O_EXCL flag, the call fails if the file already exists. In that case, the program assumes that someone else is updating the password file and exits.

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#define LOCKFILE "/etc/ptmp"
...
int pfd; /* Integer for file descriptor returned by open() call. */
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
```

Opening a File for Writing

The following example opens a file for writing, creating the file if it does not already exist. If the file exists, the system truncates the file to zero bytes.

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#define LOCKFILE "/etc/ptmp"
...
int pfd;
char filename[PATH_MAX+1];
...
if ((pfd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    perror("Cannot open output file\n"); exit(1);
}
...
```

APPLICATION USAGE

None.

RATIONALE

Except as specified in this volume of IEEE Std 1003.1-2001, the flags allowed in `o_flag` are not mutually-exclusive and any number of them may be used simultaneously.

Some implementations permit opening FIFOs with O_RDWR. Since FIFOs could be implemented in other ways, and since two file descriptors can be used to the same effect, this possibility is left as undefined.

See `getgroups()` about the group of a newly created file.
The use of `open()` to create a regular file is preferable to the use of `creat()`, because the latter is redundant and included only for historical reasons.

The use of the O_TRUNC flag on FIFOs and directories (pipes cannot be `open()`-ed) must be permissible without unexpected side effects (for example, `creat()` on a FIFO must not remove data). Since terminal special files might have type-ahead data stored in the buffer, O_TRUNC should not affect their content, particularly if a program that normally opens a regular file should open the current controlling terminal instead. Other file types, particularly implementation-defined ones, are left implementation-defined.

IEEE Std 1003.1-2001 permits [EACCES] to be returned for conditions other than those explicitly listed.

The O_NOCTTY flag was added to allow applications to avoid unintentionally acquiring a controlling terminal as a side effect of opening a terminal file. This volume of IEEE Std 1003.1-2001 does not specify how a controlling terminal is acquired, but it allows an implementation to provide this on `open()` if the O_NOCTTY flag is not set and other conditions specified in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface are met. The O_NOCTTY flag is an effective no-op if the file being opened is not a terminal device.

In historical implementations the value of O_RDONLY is zero. Because of that, it is not possible to detect the presence of O_RDONLY and another option. Future implementations should encode O_RDONLY and O_WRONLY as bit flags so that:

```
O_RDONLY | O_WRONLY == O_RDWR
```

In general, the `open()` function follows the symbolic link if `path` names a symbolic link. However, the `open()` function, when called with O_CREAT and O_EXCL, is required to fail with [EEXIST] if `path` names an existing symbolic link, even if the symbolic link refers to a nonexistent file. This behavior is required so that privileged applications can create a new file in a known location without the possibility that a symbolic link might cause the file to be created in a different location.

For example, a privileged application that must create a file with a predictable name in a user-writable directory, such as the user’s home directory, could be compromised if the user creates a symbolic link with that name that refers to a nonexistent file in a system directory. If the user can influence the contents of a file, the user could compromise the system by creating a new system configuration or spool file that would then be interpreted by the system. The test for a symbolic link which refers to a nonexistent file must be atomic with the creation of a new file.

The POSIX.1-1990 standard required that the group ID of a newly created file be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use `chown()` to set the group ID after the file is created, or determine under what conditions the implementation will set the desired group ID.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`chmod()`, `close()`, `creat()`, `dup()`, `fcntl()`, `lseek()`, `read()`, `umask()`, `unlockpt()`, `write()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<fcntl.h>`, `<sys/stat.h>`, `<sys/types.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, O_CREAT is amended to state that the group ID of the file is set to the group ID of the file’s parent directory or to the effective group ID of the process. This is a FIPS requirement.

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.

- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.

- The [ENXIO] mandatory error condition is added.

- The [EINVAL], [ENAMETOOLONG], and [ETXTBSY] optional error conditions are added.

The DESCRIPTION and ERRORS sections are updated so that items related to the optional XSI STREAMS Option Group are marked.

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

- An explanation is added of the effect of the O_CREAT and O_EXCL flags when the path refers to a symbolic link.

- The [ELOOP] optional error condition is added.

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

The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48.
NAME
opendir — open a directory

SYNOPSIS
#include <dirent.h>
DIR *opendir(const char *dirname);

DESCRIPTION
The opendir() function shall open a directory stream corresponding to the directory named by
the dirname argument. The directory stream is positioned at the first entry. If the type DIR is
implemented using a file descriptor, applications shall only be able to open up to a total of
{OPEN_MAX} files and directories.

RETURN VALUE
Upon successful completion, opendir() shall return a pointer to an object of type DIR.
Otherwise, a null pointer shall be returned and errno set to indicate the error.

ERRORS
The opendir() function shall fail if:
[EACCES] Search permission is denied for the component of the path prefix of dirname or
read permission is denied for dirname.
[ELOOP] A loop exists in symbolic links encountered during resolution of the dirname
argument.
[ENAMETOOLONG] The length of the dirname argument exceeds {PATH_MAX} or a pathname
component is longer than {NAME_MAX}.
[ENOENT] A component of dirname does not name an existing directory or dirname is an
empty string.
[ENOTDIR] A component of dirname is not a directory.

The opendir() function may fail if:
[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
resolution of the dirname argument.
[EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.
[ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the dirname
argument, the length of the substituted pathname string exceeded
{PATH_MAX}.
[ENFILE] Too many files are currently open in the system.
EXAMPLES

Open a Directory Stream

The following program fragment demonstrates how the `opendir()` function is used.

```c
#include <sys/types.h>
#include <dirent.h>
#include <libgen.h>
...
DIR *dir;
struct dirent *dp;
...
if ((dir = opendir (".")) == NULL) {
    perror ("Cannot open ".");
    exit (1);
}
while ((dp = readdir (dir)) != NULL) {
...
```

APPLICATION USAGE

The `opendir()` function should be used in conjunction with `readdir()`, `closedir()`, and `rewinddir()` to examine the contents of the directory (see the EXAMPLES section in `readdir()`). This method is recommended for portability.

RATIONALE

Based on historical implementations, the rules about file descriptors apply to directory streams as well. However, this volume of IEEE Std 1003.1-2001 does not mandate that the directory stream be implemented using file descriptors. The description of `closedir()` clarifies that if a file descriptor is used for the directory stream, it is mandatory that `closedir()` deallocate the file descriptor. When a file descriptor is used to implement the directory stream, it behaves as if the FD_CLOEXEC had been set for the file descriptor.

The directory entries for dot and dot-dot are optional. This volume of IEEE Std 1003.1-2001 does not provide a way to test a priori for their existence because an application that is portable must be written to look for (and usually ignore) those entries. Writing code that presumes that they are the first two entries does not always work, as many implementations permit them to be other than the first two entries, with a "normal" entry preceding them. There is negligible value in providing a way to determine what the implementation does because the code to deal with dot and dot-dot must be written in any case and because such a flag would add to the list of those flags (which has proven in itself to be objectionable) and might be abused.

Since the structure and buffer allocation, if any, for directory operations are defined by the implementation, this volume of IEEE Std 1003.1-2001 imposes no portability requirements for erroneous program constructs, erroneous data, or the use of unspecified values such as the use or referencing of a `dirp` value or a `dirent` structure value after a directory stream has been closed or after a `fork()` or one of the `exec` function calls.

FUTURE DIRECTIONS

None.

SEE ALSO

`closedir()`, `lstat()`, `readdir()`, `rewinddir()`, `symlink()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<dirent.h>`, `<limits.h>`, `<sys/types.h>`
opendir()

System Interfaces

CHANGE HISTORY

First released in Issue 2.

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.

- The [ELOOP] mandatory error condition is added.

- A second [ENAMETOOLONG] is added as an optional error condition.

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

- The [ELOOP] optional error condition is added.
NAME
openlog — open a connection to the logging facility

SYNOPSIS
XSI
#include <syslog.h>

void openlog(const char *ident, int logopt, int facility);

DESCRIPTION
Refer to closelog().
optarg

NAME
optarg, opterr, optind, optopt — options parsing variables

SYNOPSIS
#include <unistd.h>
extern char *optarg;
extern int opterr, optind, optopt;

DESCRIPTION
Refer to getopt().
NAME
pathconf — get configurable pathname variables

SYNOPSIS
#include <unistd.h>
long pathconf(const char *path, int name);

DESCRIPTION
Refer to `fpathconf()`.
pause()  

#include <unistd.h>

int pause(void);

The pause() function shall suspend the calling thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process, pause() shall not return.

If the action is to execute a signal-catching function, pause() shall return after the signal-catching function returns.

Since pause() suspends thread execution indefinitely unless interrupted by a signal, there is no successful completion return value. A value of -1 shall be returned and errno set to indicate the error.

The pause() function shall fail if:

A signal is caught by the calling process and control is returned from the signal-catching function.

Many common uses of pause() have timing windows. The scenario involves checking a condition related to a signal and, if the signal has not occurred, calling pause(). When the signal occurs between the check and the call to pause(), the process often blocks indefinitely. The sigprocmask() and sigsuspend() functions can be used to avoid this type of problem.

None.

None.

None.

sigsuspend(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

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

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

The APPLICATION USAGE section is added.
 NAME
 pclose — close a pipe stream to or from a process

 SYNOPSIS
 CX
 #include <stdio.h>
 int pclose(FILE *stream);

 DESCRIPTION
 The pclose() function shall close a stream that was opened by popen(), wait for the command to
terminate, and return the termination status of the process that was running the command
language interpreter. However, if a call caused the termination status to be unavailable to
pclose(), then pclose() shall return −1 with errno set to [ECHILD] to report this situation. This can
happen if the application calls one of the following functions:

 • wait()
 • waitpid() with a pid argument less than or equal to 0 or equal to the process ID of the
   command line interpreter
 • Any other function not defined in this volume of IEEE Std 1003.1-2001 that could do one of
   the above

 In any case, pclose() shall not return before the child process created by popen() has terminated.

 If the command language interpreter cannot be executed, the child termination status returned
by pclose() shall be as if the command language interpreter terminated using exit(127) or
_exit(127).

 The pclose() function shall not affect the termination status of any child of the calling process
other than the one created by popen() for the associated stream.

 If the argument stream to pclose() is not a pointer to a stream created by popen(), the result of
pclose() is undefined.

 RETURN VALUE
 Upon successful return, pclose() shall return the termination status of the command language
interpreter. Otherwise, pclose() shall return −1 and set errno to indicate the error.

 ERRORS
 The pclose() function shall fail if:

 [ECHILD] The status of the child process could not be obtained, as described above.

 EXAMPLES
 None.

 APPLICATION USAGE
 None.

 RATIONALE
 There is a requirement that pclose() not return before the child process terminates. This is
intended to disallow implementations that return [EINTR] if a signal is received while waiting.
If pclose() returned before the child terminated, there would be no way for the application to
discover which child used to be associated with the stream, and it could not do the cleanup
itself.

 If the stream pointed to by stream was not created by popen(), historical implementations of
pclose() return −1 without setting errno. To avoid requiring pclose() to set errno in this case,
IEEE Std 1003.1-2001 makes the behavior unspecified. An application should not use pclose() to
close any stream that was not created by `popen()`.

Some historical implementations of `pclose()` either block or ignore the signals SIGINT, SIGQUIT, and SIGHUP while waiting for the child process to terminate. Since this behavior is not described for the `pclose()` function in IEEE Std 1003.1-2001, such implementations are not conforming. Also, some historical implementations return [EINVAL] if a signal is received, even though the child process has not terminated. Such implementations are also considered non-conforming.

Consider, for example, an application that uses:

```c
popen("command", "r")
```

to start `command`, which is part of the same application. The parent writes a prompt to its standard output (presumably the terminal) and then reads from the `popen()`ed stream. The child reads the response from the user, does some transformation on the response (pathname expansion, perhaps) and writes the result to its standard output. The parent process reads the result from the pipe, does something with it, and prints another prompt. The cycle repeats.

Assuming that both processes do appropriate buffer flushing, this would be expected to work.

To conform to IEEE Std 1003.1-2001, `pclose()` must use `waitpid()`, or some similar function, instead of `wait()`.

The code sample below illustrates how the `pclose()` function might be implemented on a system conforming to IEEE Std 1003.1-2001.

```c
int pclose(FILE *stream)
{
    int stat;
    pid_t pid;

    pid = <pid for process created for stream by popen()>
    (void) fclose(stream);
    while (waitpid(pid, &stat, 0) == -1) {
        if (errno != EINTR) {
            stat = -1;
            break;
        }
    }
    return(stat);
}
```

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`fork()`, `popen()`, `waitpid()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`

**CHANGE HISTORY**

First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
perror — write error messages to standard error

SYNOPSIS
#include <stdio.h>

void perror(const char *s);

DESCRIPTION
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 perror() function shall map the error number accessed through the symbol errno to a language-dependent error message, which shall be written to the standard error stream as follows:

- First (if s is not a null pointer and the character pointed to by s is not the null byte), the string pointed to by s followed by a colon and a <space>.
- Then an error message string followed by a <newline>.

The contents of the error message strings shall be the same as those returned by strerror() with argument errno.

The perror() function shall mark the file associated with the standard error stream as having been written (st_ctime, st_mtime marked for update) at some time between its successful completion and exit(), abort(), or the completion of fflush() or fclose() on stderr.

The perror() function shall not change the orientation of the standard error stream.

RETURN VALUE
The perror() function shall not return a value.

ERRORS
No errors are defined.

EXAMPLES
Printing an Error Message for a Function
The following example replaces bufptr with a buffer that is the necessary size. If an error occurs, the perror() function prints a message and the program exits.

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

char *bufptr;
size_t szbuf;

if ((bufptr = malloc(szbuf)) == NULL) {
    perror("malloc"); exit(2);
}
```

APPLICATION USAGE
None.
**perror()**

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`strerror()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**
A paragraph is added to the DESCRIPTION indicating that `perror()` does not change the orientation of the standard error stream.

**Issue 6**
Extensions beyond the ISO C standard are marked.
NAME

`pipe()` — create an interprocess channel

SYNOPSIS

```c
#include <unistd.h>

int pipe(int fildes[2]);
```

DESCRIPTION

The `pipe()` function shall create a pipe and place two file descriptors, one each into the arguments `fildes[0]` and `fildes[1]`, that refer to the open file descriptions for the read and write ends of the pipe. Their integer values shall be the two lowest available at the time of the `pipe()` call. The O_NONBLOCK and FD_CLOEXEC flags shall be clear on both file descriptors. (The `fcntl()` function can be used to set both these flags.)

Data can be written to the file descriptor `fildes[1]` and read from the file descriptor `fildes[0]`. A read on the file descriptor `fildes[0]` shall access data written to the file descriptor `fildes[1]` on a first-in-first-out basis. It is unspecified whether `fildes[0]` is also open for writing and whether `fildes[1]` is also open for reading.

A process has the pipe open for reading (correspondingly writing) if it has a file descriptor open that refers to the read end, `fildes[0]` (write end, `fildes[1]`).

Upon successful completion, `pipe()` shall mark for update the `st_atime`, `st_ctime`, and `st_mtime` fields of the pipe.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned and `errno` set to indicate the error.

ERRORS

The `pipe()` function shall fail if:

- `[EMFILE]` More than `{OPEN_MAX}` minus two file descriptors are already in use by this process.
- `[ENFILE]` The number of simultaneously open files in the system would exceed a system-imposed limit.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The wording carefully avoids using the verb “to open” in order to avoid any implication of use of `open()`; see also `write()`.

FUTURE DIRECTIONS

None.

SEE ALSO

`fcntl()`, `read()`, `write()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<fcntl.h>`, `<unistd.h>`

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to indicate that certain dispositions of \texttt{fdle}[0] and \texttt{fdle}[1] are unspecified.
NAME
poll — input/output multiplexing

SYNOPSIS
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);

DESCRIPTION
The poll() function provides applications with a mechanism for multiplexing input/output over a set of file descriptors. For each member of the array pointed to by fds, poll() shall examine the given file descriptor for the event(s) specified in events. The number of pollfd structures in the fds array is specified by nfds. The poll() function shall identify those file descriptors on which an application can read or write data, or on which certain events have occurred.

The fds argument specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one member for each open file descriptor of interest. The array’s members are pollfd structures within which fd specifies an open file descriptor and events and revents are bitmasks constructed by OR’ing a combination of the following event flags:

- **POLLIN**: Data other than high-priority data may be read without blocking.
  - For STREAMS, this flag is set in revents even if the message is of zero length.
  - This flag shall be equivalent to POLLRDNORM | POLLRDBAND.

- **POLLRDNORM**: Normal data may be read without blocking.
  - For STREAMS, data on priority band 0 may be read without blocking. This flag is set in revents even if the message is of zero length.

- **POLLRDBAND**: Priority data may be read without blocking.
  - For STREAMS, data on priority bands greater than 0 may be read without blocking. This flag is set in revents even if the message is of zero length.

- **POLLPRI**: High-priority data may be read without blocking.
  - For STREAMS, this flag is set in revents even if the message is of zero length.

- **POLLOUT**: Normal data may be written without blocking.
  - For STREAMS, data on priority band 0 may be written without blocking.

- **POLLWRNORM**: Equivalent to POLLOUT.

- **POLLWRBAND**: Priority data may be written.
  - For STREAMS, data on priority bands greater than 0 may be written without blocking. If any priority band has been written to on this STREAM, this event only examines bands that have been written to at least once.

- **POLLERR**: An error has occurred on the device or stream. This flag is only valid in the revents bitmask; it shall be ignored in the events member.

- **POLLHUP**: The device has been disconnected. This event and POLLOUT are mutually-exclusive; a stream can never be writable if a hangup has occurred. However, this event and POLLIN, POLLRDNORM, POLLRDBAND, or POLLPRI are not mutually-exclusive. This flag is only valid in the revents bitmask; it shall be ignored in the events member.
POLLNVAL The specified fd value is invalid. This flag is only valid in the revents member; it shall ignored in the events member.

The significance and semantics of normal, priority, and high-priority data are file and device-specific.

If the value of fd is less than 0, events shall be ignored, and revents shall be set to 0 in that entry on return from poll().

In each pollfd structure, poll() shall clear the revents member, except that where the application requested a report on a condition by setting one of the bits of events listed above, poll() shall set the corresponding bit in revents if the requested condition is true. In addition, poll() shall set the POLLHUP, POLLERR, and POLLNVAL flag in revents if the condition is true, even if the application did not set the corresponding bit in events.

If none of the defined events have occurred on any selected file descriptor, poll() shall wait at least timeout milliseconds for an event to occur on any of the selected file descriptors. If the value of timeout is 0, poll() shall return immediately. If the value of timeout is −1, poll() shall block until a requested event occurs or until the call is interrupted.

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.

The poll() function shall not be affected by the O_NONBLOCK flag.

The poll() function shall support regular files, terminal and pseudo-terminal devices, FIFOs, pipes, sockets and STREAMS-based files. The behavior of poll() on elements of fds that refer to other types of file is unspecified.

Regular files shall always poll TRUE for reading and writing.

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.

RETURN VALUE
Upon successful completion, poll() 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 revents member is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, poll() shall return −1 and set errno to indicate the error.

ERRORS
The poll() function shall fail if:

[EAGAIN] The allocation of internal data structures failed but a subsequent request may succeed.

[EINTR] A signal was caught during poll().

[XSR][EINVAL] The nfds argument is greater than {OPEN_MAX}, or one of the fd members refers to a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.
EXAMPLES

Checking for Events on a Stream

The following example opens a pair of STREAMS devices and then waits for either one to become writable. This example proceeds as follows:

1. Sets the timeout parameter to 500 milliseconds.
2. Opens the STREAMS devices /dev/dev0 and /dev/dev1, and then polling them, specifying POLLOUT and POLLWRBAND as the events of interest.

   The STREAMS device names /dev/dev0 and /dev/dev1 are only examples of how STREAMS devices can be named; STREAMS naming conventions may vary among systems conforming to the IEEE Std 1003.1-2001.

3. Uses the ret variable to determine whether an event has occurred on either of the two STREAMS. The poll() function is given 500 milliseconds to wait for an event to occur (if it has not occurred prior to the poll() call).

4. Checks the returned value of ret. If a positive value is returned, one of the following can be done:
   a. Priority data can be written to the open STREAM on priority bands greater than 0, because the POLLWRBAND event occurred on the open STREAM (fds[0] or fds[1]).
   b. Data can be written to the open STREAM on priority-band 0, because the POLLOUT event occurred on the open STREAM (fds[0] or fds[1]).

5. If the returned value is not a positive value, permission to write data to the open STREAM (on any priority band) is denied.

6. If the POLLHUP event occurs on the open STREAM (fds[0] or fds[1]), the device on the open STREAM has disconnected.

   #include <stropts.h>
   #include <poll.h>
   ...
   struct pollfd fds[2];
   int timeout_msecs = 500;
   int ret;
   int i;

   /* Open STREAMS device. */
   fds[0].fd = open("/dev/dev0", ...);
   fds[1].fd = open("/dev/dev1", ...);
   fds[0].events = POLLOUT | POLLWRBAND;
   fds[1].events = POLLOUT | POLLWRBAND;

   ret = poll(fds, 2, timeout_msecs);

   if (ret > 0) {
      /* An event on one of the fds has occurred. */
      for (i=0; i<2; i++) {
         if (fds[i].revents & POLLWRBAND) {
            /* Priority data may be written on device number i. */
            ...
         }
      }
/* Data may be written on device number i. */
...
if (fds[i].revents & POLLHUP) {
/* A hangup has occurred on device number i. */
...
}

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.6 (on page 38), getmsg(), putmsg(), read(), select(), write(), the Base Definitions volume of IEEE Std 1003.1-2001, <poll.h>, <stropts.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
The description of POLLWRBAND is updated.

Issue 6
Text referring to sockets is added to the DESCRIPTION.
Text relating to the XSI STREAMS Option Group is marked.
The Open Group Corrigendum U055/3 is applied, updating the DESCRIPTION of POLLWRBAND.
NAME
popen — initiate pipe streams to or from a process

SYNOPSIS
#include <stdio.h>
FILE *popen(const char *command, const char *mode);

DESCRIPTION
The popen() function shall execute the command specified by the string command. It shall create a pipe between the calling program and the executed command, and shall return a pointer to a stream that can be used to either read from or write to the pipe.

The environment of the executed command shall be as if a child process were created within the popen() call using the fork() function, and the child invoked the sh utility using the call:
execl(shell path, "sh", "-c", command, (char *)0);
where shell path is an unspecified pathname for the sh utility.

The popen() function shall ensure that any streams from previous popen() calls that remain open in the parent process are closed in the new child process.

The mode argument to popen() is a string that specifies I/O mode:

1. If mode is r, when the child process is started, its file descriptor STDOUT_FILENO shall be the writable end of the pipe, and the file descriptor fileno(stream) in the calling process, where stream is the stream pointer returned by popen(), shall be the readable end of the pipe.

2. If mode is w, when the child process is started its file descriptor STDIN_FILENO shall be the readable end of the pipe, and the file descriptor fileno(stream) in the calling process, where stream is the stream pointer returned by popen(), shall be the writable end of the pipe.

3. If mode is any other value, the result is undefined.

After popen(), both the parent and the child process shall be capable of executing independently before either terminates.

Pipe streams are byte-oriented.

RETURN VALUE
Upon successful completion, popen() shall return a pointer to an open stream that can be used to read or write to the pipe. Otherwise, it shall return a null pointer and may set errno to indicate the error.

ERRORS
The popen() function may fail if:
[EMFILE] {FOPEN_MAX} or {STREAM_MAX} streams are currently open in the calling process.
[EINVAL] The mode argument is invalid.

The popen() function may also set errno values as described by fork() or pipe().
EXAMPLES

None.

APPLICATION USAGE

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.

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().

A stream opened by popen() should be closed by pclose().

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.

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().

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:

sysconf(_SC_2_VERSION)

(See sysconf()).

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 execlp() and 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)).

FUTURE DIRECTIONS

None.

SEE ALSO

pclose(), pipe(), sysconf(), system(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>, the Shell and Utilities volume of IEEE Std 1003.1-2001, sh

CHANGE HISTORY

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

Issue 5

A statement is added to the DESCRIPTION indicating that pipe streams are byte-oriented.

Issue 6

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

- The optional [EMFILE] error condition is added.
posix_fadvise()
EXAMPLES

None.

APPLICATION USAGE

The `posix_fadvise()` function is part of the Advisory Information option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`posix_madvise()`), the Base Definitions volume of IEEE Std 1003.1-2001, `<fcntl.h>`

CHANGE HISTORY


In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.
NAME
posix_fallocate — file space control (ADVANCED REALTIME)

SYNOPSIS
ADV
#include <fcntl.h>

int posix_fallocate(int fd, off_t offset, size_t len);

DESCRIPTION
The posix_fallocate() function shall ensure that any required storage for regular file data starting at offset and continuing for len bytes is allocated on the file system storage media. If posix_fallocate() returns successfully, subsequent writes to the specified file data shall not fail due to the lack of free space on the file system storage media.

If the offset+len is beyond the current file size, then posix_fallocate() shall adjust the file size to offset+len. Otherwise, the file size shall not be changed.

It is implementation-defined whether a previous posix_fadvise() call influences allocation strategy.

Space allocated via posix_fallocate() shall be freed by a successful call to creat() or open() that truncates the size of the file. Space allocated via posix_fallocate() may be freed by a successful call to ftruncate() that reduces the file size to a size smaller than offset+len.

RETURN VALUE
Upon successful completion, posix_fallocate() shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS
The posix_fallocate() function shall fail if:

[EBADF] The fd argument is not a valid file descriptor.
[EBADF] The fd argument references a file that was opened without write permission.
[EFBIG] The value of offset+len is greater than the maximum file size.
[EINTR] A signal was caught during execution.
[EINVAL] The len argument was zero or the offset argument was less than zero.
[EIO] An I/O error occurred while reading from or writing to a file system.
[ENODEV] The fd argument does not refer to a regular file.
[ENOSPC] There is insufficient free space remaining on the file system storage media.
[ESPIPE] The fd argument is associated with a pipe or FIFO.

EXAMPLES
None.

APPLICATION USAGE
The posix_fallocate() function is part of the Advisory Information option and need not be provided on all implementations.

RATIONALE
None.
posix_fallocate()

FUTURE DIRECTIONS
None.

SEE ALSO
creat(), ftruncate(), open(), unlink(), the Base Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>

CHANGE HISTORY
In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.
NAME

posix_madvise — memory advisory information and alignment control (ADVANCED
REALTIME)

SYNOPSIS

ADV
#include <sys/mman.h>

int posix_madvise(void *addr, size_t len, int advice);

DESCRIPTION

The posix_madvise() function need only be supported if either the Memory Mapped Files or the
Shared Memory Objects options are supported.

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.

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.

The advice to be applied to the memory range is specified by the advice parameter and may be
one of the following values:

POSIX_MADV_NORMAL

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.

POSIX_MADV_SEQUENTIAL

Specifies that the application expects to access the specified range sequentially from lower
addresses to higher addresses.

POSIX_MADV_RANDOM

Specifies that the application expects to access the specified range in a random order.

POSIX_MADV_WILLNEED

Specifies that the application expects to access the specified range in the near future.

POSIX_MADV_DONTNEED

Specifies that the application expects that it will not access the specified range in the near
future.

These values are defined in the <sys/mman.h> header.

RETURN VALUE

Upon successful completion, posix_madvise() shall return zero; otherwise, an error number shall
be returned to indicate the error.

ERRORS

The posix_madvise() function shall fail if:

[EINVAL] The value of advice is invalid.

[ENOMEM] Addresses in the range starting at addr and continuing for len bytes are partly
or completely outside the range allowed for the address space of the calling
process.

The posix_madvise() function may fail if:
posix_madvise()

The value of \textit{addr} is not a multiple of the value returned by \textit{sysconf()} when the name value \_SC_PAGESIZE is used.

The value of \textit{len} is zero.

EXAMPLES
None.

APPLICATION USAGE
The \textit{posix_madvise()} function is part of the Advisory Information option and need not be provided on all implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
\textit{mmap()}, \textit{posix_fadvise()}, \textit{sysconf()}, the Base Definitions volume of IEEE Std 1003.1-2001, \textless sys/mman.h\textgreater

CHANGE HISTORY
IEEE PASC Interpretation 1003.1 \#102 is applied.
NAME
posix_mem_offset — find offset and length of a mapped typed memory block (ADVANCED
REALTIME)

SYNOPSIS
#include <sys/mman.h>

int posix_mem_offset(const void *restrict addr, size_t len,
off_t *restrict off, size_t *restrict contig_len,
int *restrict fildes);

DESCRIPTION
The posix_mem_offset() function shall return in the variable pointed to by off a value that
identifies the offset (or location), within a memory object, of the memory block currently
mapped at addr. The function shall return in the variable pointed to by fildes, the descriptor used
(via mmap()) to establish the mapping which contains addr. If that descriptor was closed since
the mapping was established, the returned value of fildes shall be −1. The len argument specifies
the length of the block of the memory object the user wishes the offset for; upon return, the value
pointed to by contig_len shall equal either len, or the length of the largest contiguous block of the
memory object that is currently mapped to the calling process starting at addr, whichever is
smaller.

If the memory object mapped at addr is a typed memory object, then if the off and contig_len
values obtained by calling posix_mem_offset() are used in a call to mmap() with a file descriptor
that refers to the same memory pool as fildes (either through the same port or through a different
port), and that was opened with neither the POSIX_TYPED_MEM_ALLOCATE nor the
POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, the typed memory area that is mapped shall
be exactly the same area that was mapped at addr in the address space of the process that called
posix_mem_offset().

If the memory object specified by fildes is not a typed memory object, then the behavior of this
function is implementation-defined.

RETURN VALUE
Upon successful completion, the posix_mem_offset() function shall return zero; otherwise, the
corresponding error status value shall be returned.

ERRORS
The posix_mem_offset() function shall fail if:

[EACCES] The process has not mapped a memory object supported by this function at
the given address addr.

This function shall not return an error code of [EINVAL].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.
posix_mem_offset()

FUTURE DIRECTIONS
None.

SEE ALSO
mmap(), posix_typed_mem_open(), the Base Definitions volume of IEEE Std 1003.1-2001,
<sys/mman.h>

CHANGE HISTORY
NAME
posix_memalign — aligned memory allocation (ADVANCED REALTIME)

SYNOPSIS
ADV
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment, size_t size);

DESCRIPTION
The posix_memalign() function shall allocate size bytes aligned on a boundary specified by alignment, and shall return a pointer to the allocated memory in memptr. The value of alignment shall be a multiple of sizeof(void *), that is also a power of two. Upon successful completion, the value pointed to by memptr shall be a multiple of alignment.

The free() function shall deallocate memory that has previously been allocated by posix_memalign().

RETURN VALUE
Upon successful completion, posix_memalign() shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS
The posix_memalign() function shall fail if:

[EINVAL] The value of the alignment parameter is not a power of two multiple of sizeof(void *).

[ENOMEM] There is insufficient memory available with the requested alignment.

EXAMPLES
None.

APPLICATION USAGE
The posix_memalign() function is part of the Advisory Information option and need not be provided on all implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
free(), malloc(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY

In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.
NAME

posix_openpt — open a pseudo-terminal device

SYNOPSIS

XSI
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(int oflag);

DESCRIPTION

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.

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>:

O_RDWR  Open for reading and writing.
O_NOCTTY If set posix_openpt() shall not cause the terminal device to become the
controlling terminal for the process.

The behavior of other values for the oflag argument is unspecified.

RETURN VALUE

Upon successful completion, the posix_openpt() function shall open a master pseudo-terminal
dervice and return a non-negative integer representing the lowest numbered unused file
description. Otherwise, −1 shall be returned and errno set to indicate the error.

ERRORS

The posix_openpt() function shall fail if:

[EMFILE]  {OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]  The maximum allowable number of files is currently open in the system.

The posix_openpt() function may fail if:

[EINVAL]  The value of oflag is not valid.
[EAGAIN]  Out of pseudo-terminal resources.

[XSR][ENOSR]  Out of STREAMS resources.

EXAMPLES

Opening a Pseudo-Terminal and Returning the Name of the Slave Device and a File Descriptor

#include <fcntl.h>
#include <stdio.h>

int masterfd, slavefd;
char *slavedevice;

masterfd = posix_openpt(O_RDWR|O_NOCTTY);
if (masterfd == -1
    || grantpt (masterfd) == -1

870  System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
|| unlockpt (masterfd) == -1
|| (slavedevice = ptsname (masterfd)) == NULL)
return -1;

printf("slave device is: %s\n", slavedevice);
slavefd = open(slave, O_RDWR|O_NOCTTY);
if (slavefd < 0)
    return -1;

APPLICATION USAGE
This function is a method for portably obtaining a file descriptor of a master terminal device for
a pseudo-terminal. The grantpt() and ptsname() functions can be used to manipulate mode and
ownership permissions, and to obtain the name of the slave device, respectively.

RATIONALE
The standard developers considered the matter of adding a special device for cloning master
pseudo-terminals: the /dev/ptmx device. However, consensus could not be reached, and it was
felt that adding a new function would permit other implementations. The posix_openpt() function is designed to complement the grantpt(), ptsname(), and unlockpt() functions.

On implementations supporting the /dev/ptmx clone device, opening the master device of a pseudo-terminal is simply:

mfds = open("/dev/ptmx", oflag );
if (mfds < 0)
    return -1;

FUTURE DIRECTIONS
None.

SEE ALSO
grantpt(), open(), ptsname(), unlockpt(), the Base Definitions volume of IEEE Std 1003.1-2001,
<fcntl.h>

CHANGE HISTORY
First released in Issue 6.
NAME
posix_spawn, posix_spawnp — spawn a process (ADVANCED REALTIME)

SYNOPSIS
#include <spawn.h>

int posix_spawn(pid_t *restrict pid, const char *restrict path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char *const envp[restrict]);

int posix_spawnp(pid_t *restrict pid, const char *restrict file,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char * const envp[restrict]);

DESCRIPTION
The posix_spawn() and posix_spawnp() functions shall create a new process (child process) from
the specified process image. The new process image shall be constructed from a regular
executable file called the new process image file.

When a C program is executed as the result of this call, it shall be entered as a C-language
function call as follows:

    int main(int argc, char *argv[]);

where argc is the argument count and argv is an array of character pointers to the arguments
themselves. In addition, the following variable:

    extern char **environ;

shall be initialized as a pointer to an array of character pointers to the environment strings.

The argument argv is an array of character pointers to null-terminated strings. The last member
of this array shall be a null pointer and is not counted in argc. These strings constitute the
argument list available to the new process image. The value in argv[0] should point to a filename
that is associated with the process image being started by the posix_spawn() or posix_spawnp()
function.

The argument envp is an array of character pointers to null-terminated strings. These strings
constitute the environment for the new process image. The environment array is terminated by a
null pointer.

The number of bytes available for the child process’ combined argument and environment lists
is {ARG_MAX}. The implementation shall specify in the system documentation (see the Base
Definitions volume of IEEE Std 1003.1-2001, Chapter 2, Conformance) whether any list
overhead, such as length words, null terminators, pointers, or alignment bytes, is included in
this total.

The path argument to posix_spawn() is a pathname that identifies the new process image file to
execute.

The file parameter to posix_spawnp() shall be used to construct a pathname that identifies the
new process image file. If the file parameter contains a slash character, the file parameter shall be
used as the pathname for the new process image file. Otherwise, the path prefix for this file shall
be 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 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 `fcntl()`). For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks (see `fcntl()`), 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 `fcntl()`), 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 `fcntl()`) 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
of the new process image file is set, the effective group ID of the child process shall become that
file's group ID before the new process image begins execution.

If the POSIX_SPAWN_SETSIGMASK flag is set in the spawn-flags attribute of the object
referenced by attrp, the child process shall initially have the signal mask specified in the spawn-
sigmask attribute of the object referenced by attrp.

If the POSIX_SPAWN_SETSIGDEF flag is set in the spawn-flags attribute of the object referenced
by attrp, the signals specified in the spawn-sigdefault attribute of the same object shall be set to
their default actions in the child process. Signals set to the default action in the parent process
shall be set to the default action in the child process.

Signals set to be caught by the calling process shall be set to the default action in the child
process.

Except for SIGCHLD, signals set to be ignored by the calling process image shall be set to be
ignored by the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag
being set in the spawn-flags attribute of the object referenced by attrp and the signals being
indicated in the spawn-sigdefault attribute of the object referenced by attrp.

If the SIGCHLD signal is set to be ignored by the calling process, it is unspecified whether the
SIGCHLD signal is set to be ignored or to the default action in the child process, unless
otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the spawn_flags
attribute of the object referenced by attrp and the SIGCHLD signal being indicated in the
spawn_sigdefault attribute of the object referenced by attrp.

If the value of the attrp pointer is NULL, then the default values are used.

All process attributes, other than those influenced by the attributes set in the object referenced
by attrp as specified above or by the file descriptor manipulations specified in file_actions, shall
appear in the new process image as though fork() had been called to create a child process and
then a member of the exec family of functions had been called by the child process to execute the
new process image.

THR It is implementation-defined whether the fork handlers are run when posix_spawn() or
posix_spawnp() is called.

RETURN VALUE
Upon successful completion, posix_spawn() and posix_spawnp() shall return the process ID of the
child process to the parent process, in the variable pointed to by a non-NULL pid argument, and
shall return zero as the function return value. Otherwise, no child process shall be created, the
value stored into the variable pointed to by a non-NULL pid is unspecified, and an error number
shall be returned as the function return value to indicate the error. If the pid argument is a null
pointer, the process ID of the child is not returned to the caller.

ERRORS
The posix_spawn() and posix_spawnp() functions may fail if:

[EINVAL] The value specified by file_actions or attrp is invalid.

If this error occurs after the calling process successfully returns from the posix_spawn() or
posix_spawnp() function, the child process may exit with exit status 127.

If posix_spawn() or posix_spawnp() fail for any of the reasons that would cause fork() or one of
the exec family of functions to fail, an error value shall be returned as described by fork() and
exec, respectively (or, if the error occurs after the calling process successfully returns, the child
process shall exit with exit status 127).
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 error value shall be returned as described by setpgid() (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If POSIX_SPAWN_SETSCHEDPARAM is set and POSIX_SPAWN_SETSCHEDULER is not set in the spawn-flags attribute of the object referenced by attrp, then if posix_spawn() or posix_spawnp() fails for any of the reasons that would cause sched_setparam() to fail, an error value shall be returned as described by sched_setparam() (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If POSIX_SPAWN_SETSCHEDULER is set in the spawn-flags attribute of the object referenced by attrp, and if posix_spawn() or posix_spawnp() fails for any of the reasons that would cause sched_setscheduler() to fail, an error value shall be returned as described by sched_setscheduler() (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If the file_actions argument is not NULL, and specifies any close, dup2, or open actions to be performed, and if posix_spawn() or posix_spawnp() fails for any of the reasons that would cause close(), dup2(), or open() to fail, an error value shall be returned as described by close(), dup2(), and open(), respectively (or, if the error occurs after the calling process successfully returns, the 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().

EXAMPLES
None.

APPLICATION USAGE
These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE
The posix_spawn() function and its close relation posix_spawnp() have been introduced to overcome the following perceived difficulties with fork(): the fork() function is difficult or impossible to implement without swapping or dynamic address translation.

- Swapping is generally too slow for a realtime environment.
- Dynamic address translation is not available everywhere that POSIX might be useful.
- Processes are too useful to simply option out of POSIX whenever it must run without address translation or other MMU services.

Thus, POSIX needs process creation and file execution primitives that can be efficiently implemented without address translation or other MMU services.

The posix_spawn() function is implementable as a library routine, but both posix_spawn() and posix_spawnp() are designed as kernel operations. Also, although they may be an efficient replacement for many fork()/exec pairs, their goal is to provide useful process creation primitives for systems that have difficulty with fork(), not to provide drop-in replacements for fork()/exec.

This view of the role of posix_spawn() and posix_spawnp() influenced the design of their API. It does not attempt to provide the full functionality of fork()/exec in which arbitrary user-specified operations of any sort are permitted between the creation of the child process and the execution of the new process image; any attempt to reach that level would need to provide a programming language as parameters. Instead, posix_spawn() and posix_spawnp() are process creation primitives like the Start_Process and Start_Process_Search Ada language bindings package POSIX_Process_Primitives and also like those in many operating systems that are not UNIX
To achieve its coverage goals, `posix_spawn()` and `posix_spawnp()` have control of six types of
inheritance: file descriptors, process group ID, user and group ID, signal mask, scheduling, and
whether each signal ignored in the parent will remain ignored in the child, or be reset to its
default action in the child.

Control of file descriptors is required to allow an independently written child process image to
access data streams opened by and even generated or read by the parent process without being
specifically coded to know which parent files and file descriptors are to be used. Control of the
process group ID is required to control how the child process' job control relates to that of the
parent.

Control of the signal mask and signal defaulting is sufficient to support the implementation of
`system()`. Although support for `system()` is not explicitly one of the goals for `posix_spawn()` and
`posix_spawnp()`, it is covered under the “at least 50%” coverage goal.

The intention is that the normal file descriptor inheritance across `fork()`, the subsequent effect of
the specified spawn file actions, and the normal file descriptor inheritance across one of the `exec`
family of functions should fully specify open file inheritance. The implementation need make no
decisions regarding the set of open file descriptors when the child process image begins
execution, those decisions having already been made by the caller and expressed as the set of
open file descriptors and their `FD_CLOEXEC` flags at the time of the call and the spawn file
actions object specified in the call. We have been assured that in cases where the POSIX
`Start_Process` Ada primitives have been implemented in a library, this method of controlling file
descriptor inheritance may be implemented very easily.

We can identify several problems with `posix_spawn()` and `posix_spawnp()`, but there does not
appear to be a solution that introduces fewer problems. Environment modification for child
process attributes not specifiable via the `attrp` or `file_actions` arguments must be done in the
parent process, and since the parent generally wants to save its context, it is more costly than
similar functionality with `fork()`/`exec`. It is also complicated to modify the environment of a
multi-threaded process temporarily, since all threads must agree when it is safe for the
environment to be changed. However, this cost is only borne by those invocations of
`posix_spawn()` and `posix_spawnp()` that use the additional functionality. Since extensive
modifications are not the usual case, and are particularly unlikely in time-critical code, keeping
much of the environment control out of `posix_spawn()` and `posix_spawnp()` is appropriate design.

The `posix_spawn()` and `posix_spawnp()` functions do not have all the power of `fork()`/`exec`. This is
to be expected. The `fork()` function is a wonderfully powerful operation. We do not expect to
duplicate its functionality in a simple, fast function with no special hardware requirements. It is
worth noting that `posix_spawn()` and `posix_spawnp()` are very similar to the process creation
operations on many operating systems that are not UNIX systems.

**Requirements**

The requirements for `posix_spawn()` and `posix_spawnp()` are:

- They must be implementable without an MMU or unusual hardware.
- They must be compatible with existing POSIX standards.

Additional goals are:

- They should be efficiently implementable.
- They should be able to replace at least 50% of typical executions of `fork()`.
A system with `posix_spawn()` and `posix_spawnp()` and without `fork()` should be useful, at least for realtime applications.

A system with `fork()` and the `exec` family should be able to implement `posix_spawn()` and `posix_spawnp()` as library routines.

**Two-Syntax**

POSIX `exec` has several calling sequences with approximately the same functionality. These appear to be required for compatibility with existing practice. Since the existing practice for the `posix_spawn*()` functions is otherwise substantially unlike POSIX, we feel that simplicity outweighs compatibility. There are, therefore, only two names for the `posix_spawn*()` functions.

The parameter list does not differ between `posix_spawn()` and `posix_spawnp()`; `posix_spawnp()` interprets the second parameter more elaborately than `posix_spawn()`.

**Compatibility with POSIX.5 (Ada)**

The `Start_Process` and `Start_Process_Search` procedures from the `POSIX_Process_Primitives` package from the Ada language binding to POSIX.1 encapsulate `fork()` and `exec` functionality in a manner similar to that of `posix_spawn()` and `posix_spawnp()`. Originally, in keeping with our simplicity goal, the standard developers had limited the capabilities of `posix_spawn()` and `posix_spawnp()` to a subset of the capabilities of `Start_Process` and `Start_Process_Search`; certain non-default capabilities were not supported. However, based on suggestions by the ballot group to improve file descriptor mapping or drop it, and on the advice of an Ada Language Bindings working group member, the standard developers decided that `posix_spawn()` and `posix_spawnp()` should be sufficiently powerful to implement `Start_Process` and `Start_Process_Search`. The rationale is that if the Ada language binding to such a primitive had already been approved as an IEEE standard, there can be little justification for not approving the functionally-equivalent parts of a C binding. The only three capabilities provided by `posix_spawn()` and `posix_spawnp()` that are not provided by `Start_Process` and `Start_Process_Search` are optionally specifying the child’s process group ID, the set of signals to be reset to default signal handling in the child process, and the child’s scheduling policy and parameters.

For the Ada language binding for `Start_Process` to be implemented with `posix_spawn()`, that binding would need to explicitly pass an empty signal mask and the parent’s environment to `posix_spawn()` whenever the caller of `Start_Process` allowed these arguments to default, since `posix_spawn()` does not provide such defaults. The ability of `Start_Process` to mask user-specified signals during its execution is functionally unique to the Ada language binding and must be dealt with in the binding separately from the call to `posix_spawn()`.

**Process Group**

The process group inheritance field can be used to join the child process with an existing process group. By assigning a value of zero to the `spawn-pgroup` attribute of the object referenced by `attrp`, the `setpgid()` mechanism will place the child process in a new process group.
Threads

Without the `posix_spawn()` and `posix_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.

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.

Asynchronous Error Notification

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()`.

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.

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.

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.

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...
of the child process before the new process image executes are reported by setting the child’s exit status to 127. The calling process may use the WIFEXITED and WEXITSTATUS macros on the stat_val stored by the wait() or waitpid() functions to detect spawn failures to the extent that other status values with which the child process image may exit (before the parent can conclusively determine that the child process image has begun execution) are distinct from exit status 127.

FUTURE DIRECTIONS
None.

SEE ALSO
alarm(), chmod(), close(), dup(), exec(), exit(), fcntl(), fork(), kill(), open(), posix_spawn_file_actions_addclose(), posix_spawn_file_actions_adddup2(), posix_spawn_file_actions_addopen(), posix_spawn_file_actions_destroy(), <REFERENCE UNDEFINED>(posix_spawn_file_actions_init), posix_spawnattr_destroy(), posix_spawnattr_init(), posix_spawnattr_getsigdefault(), posix_spawnattr_getflags(), posix_spawnattr_getpgroup(), posix_spawnattr_getschedparam(), posix_spawnattr_getschedpolicy(), posix_spawnattr_getsigmask(), posix_spawnattr_getsigsetdefault(), posix_spawnattr_getflags(), posix_spawnattr_getpgroup(), posix_spawnattr_getschedparam(), posix_spawnattr_getschedpolicy(), posix_spawnattr_getsigmask(), sched_setparam(), sched_setscheduler(), setpgid(), setuid(), stat(), times(), wait(), the Base Definitions volume of IEEE Std 1003.1-2001, <spawn.h>

CHANGE HISTORY
IEEE PASC Interpretation 1003.1 #103 is applied, noting that the signal default actions are changed as well as the signal mask in step 2.
IEEE PASC Interpretation 1003.1 #132 is applied.
NAME
posix_spawn_file_actions_addclose, posix_spawn_file_actions_addopen — add close or open action to spawn file actions object (ADVANCED REALTIME)

SYNOPSIS
#include <spawn.h>

int posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *file_actions, int fildes);
int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *file_actions, int fildes, const char *restrict path, int oflag, mode_t mode);

DESCRIPTION
These functions shall add or delete a close or open action to a spawn file actions object.

A spawn file actions object is of type posix_spawn_file_actions_t (defined in <spawn.h>) and is used to specify a series of actions to be performed by a posix_spawn() or posix_spawnp() operation in order to arrive at the set of open file descriptors for the child process given the set of open file descriptors of the parent. IEEE Std 1003.1-2001 does not define comparison or assignment operators for the type posix_spawn_file_actions_t.

A spawn file actions object, when passed to posix_spawn() or posix_spawnp(), shall specify how the set of open file descriptors in the calling process is transformed into a set of potentially open file descriptors for the spawned process. This transformation shall be as if the specified sequence of actions was performed exactly once, in the context of the spawned process (prior to execution of the new process image), in the order in which the actions were added to the object; additionally, when the new process image is executed, any file descriptor (from this new set) which has its FD_CLOEXEC flag set shall be closed (see posix_spawn()).

The posix_spawn_file_actions_addclose() function shall add a close action to the object referenced by file_actions that shall cause the file descriptor fildes to be closed (as if close(fildes) had been called) when a new process is spawned using this file actions object.

The posix_spawn_file_actions_addopen() function shall add an open action to the object referenced by file_actions that shall cause the file named by path to be opened (as if open(path, oflag, mode) had been called, and the returned file descriptor, if not fildes, had been changed to fildes) when a new process is spawned using this file actions object. If fildes was already an open file descriptor, it shall be closed before the new file is opened.

The string described by path shall be copied by the posix_spawn_file_actions_addopen() function.

RETURN VALUE
Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS
These functions shall fail if:

[EBADF] The value specified by fildes is negative or greater than or equal to [OPEN_MAX].

These functions may fail if:

[EINVAL] The value specified by file_actions is invalid.

[ENOMEM] Insufficient memory exists to add to the spawn file actions object.
It shall not be considered an error for the *fildes* argument passed to these functions to specify a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is later used during a *posix_spawn()* or *posix_spawnp()* operation.

**EXAMPLES**

None.

**APPLICATION USAGE**

These functions are part of the Spawn option and need not be provided on all implementations.

**RATIONALE**

A spawn file actions object may be initialized to contain an ordered sequence of *close()*), *dup2()*), and *open()* operations to be used by *posix_spawn()* or *posix_spawnp()* to arrive at the set of open file descriptors inherited by the spawned process from the set of open file descriptors in the parent at the time of the *posix_spawn()* or *posix_spawnp()* call. It had been suggested that the *close()* and *dup2()* operations alone are sufficient to rearrange file descriptors, and that files which need to be opened for use by the spawned process can be handled either by having the calling process open them before the *posix_spawn()* or *posix_spawnp()* call (and close them after), or by passing filenames to the spawned process (in *argv*) so that it may open them itself. The standard developers recommend that applications use one of these two methods when practical, since detailed error status on a failed open operation is always available to the application this way. However, the standard developers feel that allowing a spawn file actions object to specify open operations is still appropriate because:

1. It is consistent with equivalent POSIX.5 (Ada) functionality.
2. It supports the I/O redirection paradigm commonly employed by POSIX programs designed to be invoked from a shell. When such a program is the child process, it may not be designed to open files on its own.
3. It allows file opens that might otherwise fail or violate file ownership/access rights if executed by the parent process.

Regarding 2. above, note that the spawn open file action provides to *posix_spawn()* and *posix_spawnp()* the same capability that the shell redirection operators provide to *system()*), only without the intervening execution of a shell; for example:

```
    system ("myprog <file1 3<file2");
```

Regarding 3. above, note that if the calling process needs to open one or more files for access by the spawned process, but has insufficient spare file descriptors, then the open action is necessary to allow the *open()* to occur in the context of the child process after other file descriptors have been closed (that must remain open in the parent).

Additionally, if a parent is executed from a file having a “set-user-id” mode bit set and the POSIX_SPAWN_RESETIDS flag is set in the spawn attributes, a file created within the parent process will (possibly incorrectly) have the parent’s effective user ID as its owner, whereas a file created via an *open()* action during *posix_spawn()* or *posix_spawnp()* will have the parent’s real ID as its owner; and an open by the parent process may successfully open a file to which the real user should not have access or fail to open a file to which the real user should have access.
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.

It was noted by a member of the Ada Language Bindings working group that the approved Ada Language `Start_Process` family of POSIX process primitives use a caller-specified set of file actions to alter the normal `fork()`/`exec` semantics for inheritance of file descriptors in a very flexible way, yet no such problems exist because the burden of determining how to achieve the final file descriptor mapping is completely on the application. Furthermore, although the file actions interface appears frightening at first glance, it is actually quite simple to implement in either a library or the kernel.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`close()`, `dup()`, `open()`, `posix_spawn()`, `posix_spawn_file_actions_adddup2()`, `posix_spawn_file_actions_destroy()`, `posix_spawnp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<spawn.h>`

**CHANGE HISTORY**


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.
NAME
posix_spawn_file_actions_adddup2 — add dup2 action to spawn file actions object
(ADVANCED REALTIME)

SYNOPSIS
#include <spawn.h>

int posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t * file_actions, int fildes, int newfildes);

DESCRIPTION
The posix_spawn_file_actions_adddup2() function shall add a dup2() action to the object
referenced by file_actions that shall cause the file descriptor fildes to be duplicated as newfildes (as
if dup2(fildes, newfildes) had been called) when a new process is spawned using this file actions
object.

A spawn file actions object is as defined in posix_spawn_file_actions_addclose().

RETURN VALUE
Upon successful completion, the posix_spawn_file_actions_adddup2() function shall return zero;
otherwise, an error number shall be returned to indicate the error.

ERRORS
The posix_spawn_file_actions_adddup2() function shall fail if:

[EBADF] The value specified by fildes or newfildes is negative or greater than or equal to
[OPEN_MAX].

[ENOMEM] Insufficient memory exists to add to the spawn file actions object.

The posix_spawn_file_actions_adddup2() function may fail if:

[EINVAL] The value specified by file_actions is invalid.

It shall not be considered an error for the fildes argument passed to the
posix_spawn_file_actions_adddup2() function to specify a file descriptor for which the specified
operation could not be performed at the time of the call. Any such error will be detected when
the associated file actions object is later used during a posix_spawn() or posix_spawnp() operation.

EXAMPLES
None.

APPLICATION USAGE
The posix_spawn_file_actions_adddup2() function is part of the Spawn option and need not be
provided on all implementations.

RATIONALE
Refer to the RATIONALE in posix_spawn_file_actions_addclose().

FUTURE DIRECTIONS
None.

SEE ALSO
dup(), posix_spawn(), posix_spawn_file_actions_addclose(), posix_spawn_file_actions_destroy(),
posix_spawnp(), the Base Definitions volume of IEEE Std 1003.1-2001, <spawn.h>

IEEE PASC Interpretation 1003.1 #104 is applied, noting that the [EBADF] error can apply to the newfildes argument in addition to fildes.
NAME
posix_spawn_file_actions_addopen — add open action to spawn file actions object
(ADVANCED REALTIME)

SYNOPSIS

```
#include <spawn.h>

int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *
    restrict file_actions, int fildes,
    const char *restrict path, int oflag, mode_t mode);
```

DESCRIPTION

Refer to `posix_spawn_file_actions_addclose()`.
NAME
posix_spawn_file_actions_destroy, posix_spawn_file_actions_init — destroy and initialize
spawn file actions object (ADVANCED REALTIME)

SYNOPSIS
#include <spawn.h>

int posix_spawn_file_actions_destroy(posix_spawn_file_actions_t *
    file_actions);

int posix_spawn_file_actions_init(posix_spawn_file_actions_t *
    file_actions);

DESCRIPTION
The posix_spawn_file_actions_destroy() function shall destroy the object referenced by file_actions;
the object becomes, in effect, uninitialized. An implementation may cause
posix_spawn_file_actions_destroy() to set the object referenced by file_actions to an invalid value. A
destroyed spawn file actions object can be reinitialized using posix_spawn_file_actions_init(); the
results of otherwise referencing the object after it has been destroyed are undefined.

The posix_spawn_file_actions_init() function shall initialize the object referenced by file_actions to
contain no file actions for posix_spawn() or posix_spawnp() to perform.

A spawn file actions object is as defined in posix_spawn_file_actions_addclose().

The effect of initializing an already initialized spawn file actions object is undefined.

RETURN VALUE
Upon successful completion, these functions shall return zero; otherwise, an error number shall
be returned to indicate the error.

ERRORS
The posix_spawn_file_actions_init() function shall fail if:

[ENOMEM] Insufficient memory exists to initialize the spawn file actions object.

The posix_spawn_file_actions_destroy() function may fail if:

[EINVAL] The value specified by file_actions is invalid.

EXAMPLES
None.

APPLICATION USAGE
These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE
Refer to the RATIONALE in posix_spawn_file_actions_addclose().

FUTURE DIRECTIONS
None.

SEE ALSO
posix_spawn(), posix_spawnp(), the Base Definitions volume of IEEE Std 1003.1-2001, <spawn.h>

CHANGE HISTORY

In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.
NAME
posix_spawnattr_destroy, posix_spawnattr_init — destroy and initialize spawn attributes object (ADVANCED REALTIME)

SYNOPSIS
SPN
#include <spawn.h>

int posix_spawnattr_destroy(posix_spawnattr_t *attr);
int posix_spawnattr_init(posix_spawnattr_t *attr);

DESCRIPTION
The posix_spawnattr_destroy() function shall destroy a spawn attributes object. A destroyed attr attributes object can be reinitialized using posix_spawnattr_init(); the results of otherwise referencing the object after it has been destroyed are undefined. An implementation may cause posix_spawnattr_destroy() to set the object referenced by attr to an invalid value.

The posix_spawnattr_init() function shall initialize a spawn attributes object attr with the default value for all of the individual attributes used by the implementation. Results are undefined if posix_spawnattr_init() is called specifying an already initialized attr attributes object.

A spawn attributes object is of type posix_spawnattr_t (defined in <spawn.h>) and is used to specify the inheritance of process attributes across a spawn operation. IEEE Std 1003.1-2001 does not define comparison or assignment operators for the type posix_spawnattr_t.

Each implementation shall document the individual attributes it uses and their default values unless these values are defined by IEEE Std 1003.1-2001. Attributes not defined by IEEE Std 1003.1-2001, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

The resulting spawn attributes object (possibly modified by setting individual attribute values), is used to modify the behavior of posix_spawn() or posix_spawnp(). After a spawn attributes object has been used to spawn a process by a call to a posix_spawn() or posix_spawnp(), any function affecting the attributes object (including destruction) shall not affect any process that has been spawned in this way.

RETURN VALUE
Upon successful completion, posix_spawnattr_destroy() and posix_spawnattr_init() shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS
The posix_spawnattr_init() function shall fail if:
[ENOMEM] Insufficient memory exists to initialize the spawn attributes object.

The posix_spawnattr_destroy() function may fail if:
[EINVAL] The value specified by attr is invalid.

EXAMPLES
None.

APPLICATION USAGE
These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE
The original spawn interface proposed in IEEE Std 1003.1-2001 defined the attributes that specify the inheritance of process attributes across a spawn operation as a structure. In order to be able to separate optional individual attributes under their appropriate options (that is, the spawn-schedparam and spawn-schedpolicy attributes depending upon the Process Scheduling option), and
also for extensibility and consistency with the newer POSIX interfaces, the attributes interface
has been changed to an opaque data type. This interface now consists of the type

posix_spawnattr_t, representing a spawn attributes object, together with associated functions to
initialize or destroy the attributes object, and to set or get each individual attribute. Although the
new object-oriented interface is more verbose than the original structure, it is simple to use,
more extensible, and easy to implement.

FUTURE DIRECTIONS
None.

SEE ALSO
posix_spawn(), posix_spawnattr_getsigdefault(), posix_spawnattr_getflags(),
posix_spawnattr_getpgroup(), posix_spawnattr_getschedparam(), posix_spawnattr_getschedpolicy(),
posix_spawnattr_getsigmask(), posix_spawnattr_setsigdefault(), posix_spawnattr_setflags(),
posix_spawnattr_setsiggroup(), posix_spawnattr_setsigmask(), posix_spawnattr_setschedpolicy(),
posix_spawnattr_setschedparam(), posix_spawnp(), the Base Definitions volume of
IEEE Std 1003.1-2001, <spawn.h>

CHANGE HISTORY
IEEE PASC Interpretation 1003.1 #106 is applied, noting that the effect of initializing an already
initialized spawn attributes option is undefined.
NAME
posix_spawnattr_getflags(), posix_spawnattr_setflags — get and set the spawn-flags attribute of a
spawn attributes object (ADVANCED REALTIME)

SYNOPSIS
#include <spawn.h>

int posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
    short *restrict flags);
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);

DESCRIPTION
The posix_spawnattr_getflags() function shall obtain the value of the spawn-flags attribute from
the attributes object referenced by attr.

The posix_spawnattr_setflags() function shall set the spawn-flags attribute in an initialized
attributes object referenced by attr.

The spawn-flags attribute is used to indicate which process attributes are to be changed in the
new process image when invoking posix_spawn() or posix_spawnp(). It is the bitwise-inclusive
OR of zero or more of the following flags:

POSIX_SPAWN_RESETIDS
POSIX_SPAWN_SETPGROUP
POSIX_SPAWN_SETSIGDEF
POSIX_SPAWN_SETSIGMASK
POSIX_SPAWN_SETSCHEDPARAM
POSIX_SPAWN_SETSCHEDULER

These flags are defined in <spawn.h>. The default value of this attribute shall be as if no flags
were set.

RETURN VALUE
Upon successful completion, posix_spawnattr_getflags() shall return zero and store the value of
the spawn-flags attribute of attr into the object referenced by the flags parameter; otherwise, an
error number shall be returned to indicate the error.

Upon successful completion, posix_spawnattr_setflags() shall return zero; otherwise, an error
number shall be returned to indicate the error.

ERRORS
These functions may fail if:

[EINVAL] The value specified by attr is invalid.

The posix_spawnattr_setflags() function may fail if:

[EINVAL] The value of the attribute being set is not valid.
EXEMPLARY

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn(), posix_spawnattr_destroy(), posix_spawnattr_init(), posix_spawnattr_getsigdefault(),
posix_spawnattr_getpgroup(), posix_spawnattr_getschedparam(), posix_spawnattr_getschedpolicy(),
posix_spawnattr_getsigmask(), posix_spawnattr_setsigdefault(), posix_spawnattr_setpgroup(),
posix_spawnattr_setschedparam(), posix_spawnattr_setschedpolicy(), posix_spawnattr_setsigmask(),
posix_spawnp(), the Base Definitions volume of IEEE Std 1003.1-2001, <spawn.h>

CHANGE HISTORY

NAME

posix_spawnattr_getpgroup, posix_spawnattr_setpgroup — get and set the spawn-pgroup attribute of a spawn attributes object (ADVANCED REALTIME)

SYNOPSIS

```c
#include <spawn.h>

int posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr, 
                               pid_t *restrict pgroup);
int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

DESCRIPTION

The `posix_spawnattr_getpgroup()` function shall obtain the value of the `spawn-pgroup` attribute from the attributes object referenced by `attr`.

The `posix_spawnattr_setpgroup()` function shall set the `spawn-pgroup` attribute in an initialized attributes object referenced by `attr`.

The `spawn-pgroup` attribute represents the process group to be joined by the new process image in a spawn operation (if POSIX_SPAWN_SETPGROUP is set in the `spawn-flags` attribute). The default value of this attribute shall be zero.

RETURN VALUE

Upon successful completion, `posix_spawnattr_getpgroup()` shall return zero and store the value of the `spawn-pgroup` attribute of `attr` into the object referenced by the `pgroup` parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, `posix_spawnattr_setpgroup()` shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

- [EINVAL] The value specified by `attr` is invalid.

The `posix_spawnattr_setpgroup()` function may fail if:

- [EINVAL] The value of the attribute being set is not valid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`posix_spawn()`, `posix_spawnattr_destroy()`, `posix_spawnattr_init()`, `posix_spawnattr_getsigdefault()`, `posix_spawnattr_getflags()`, `posix_spawnattr_getschedparam()`, `posix_spawnattr_getschedpolicy()`, `posix_spawnattr_getsigmask()`, `posix_spawnattr_setsigdefault()`, `posix_spawnattr_setsigmask()`, `posix_spawnattr_setschedparam()`, `posix_spawnattr_setschedpolicy()`, `posix_spawnattr_setsigmask()`, `posix_spawnp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<spawn.h>`
posix_spawnattr_getpgroup()

CHANGE HISTORY
NAME

posix_spawnattr_getschedparam, posix_spawnattr_setschedparam — get and set the spawn-schedparam attribute of a spawn attributes object (ADVANCED REALTIME)

SYNOPSIS

SPN PS

```c
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedparam(const posix_spawnattr_t * restrict attr, struct sched_param *restrict schedparam);
int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
        const struct sched_param *restrict schedparam);
```

DESCRIPTION

The `posix_spawnattr_getschedparam()` function shall obtain the value of the spawn-schedparam attribute from the attributes object referenced by `attr`.

The `posix_spawnattr_setschedparam()` function shall set the spawn-schedparam attribute in an initialized attributes object referenced by `attr`.

The spawn-schedparam attribute represents the scheduling parameters to be assigned to the new process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER or POSIX_SPAWN_SETSCHEDPARAM is set in the spawn-flags attribute). The default value of this attribute is unspecified.

RETURN VALUE

Upon successful completion, `posix_spawnattr_getschedparam()` shall return zero and store the value of the spawn-schedparam attribute of `attr` into the object referenced by the schedparam parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, `posix_spawnattr_setschedparam()` shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

- [EINVAL] The value specified by `attr` is invalid.

The `posix_spawnattr_setschedparam()` function may fail if:

- [EINVAL] The value of the attribute being set is not valid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn and Process Scheduling options and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`posix_spawn()`, `posix_spawnattr_destroy()`, `posix_spawnattr_init()`, `posix_spawnattr_getsigdefault()`, `posix_spawnattr_getflags()`, `posix_spawnattr_getgroup()`, `posix_spawnattr_getschedpolicy()`, `posix_spawnattr_getsigmask()`, `posix_spawnattr_setsigdefault()`, `posix_spawnattr_setflags()`,
posix_spawnattr_getschedparam()

**System Interfaces**

29019  `posix_spawnattr_setpgroup()`, `posix_spawnattr_setschedpolicy()`, `posix_spawnattr_setsigmask()`,
29020  `posix_spawnp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sched.h>`, `<spawn.h>`

29021 **CHANGE HISTORY**
NAME
posix_spawnattr_getschedpolicy(), posix_spawnattr_setschedpolicy — get and set the
spawn-schedpolicy attribute of a spawn attributes object (ADVANCED REALTIME)

SYNOPSIS
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedpolicy(const posix_spawnattr_t * attr, int *restrict schedpolicy);
int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr, int schedpolicy);

DESCRIPTION
The posix_spawnattr_getschedpolicy() function shall obtain the value of the spawn-schedpolicy
attribute from the attributes object referenced by attr.

The posix_spawnattr_setschedpolicy() function shall set the spawn-schedpolicy attribute in an
initialized attributes object referenced by attr.

The spawn-schedpolicy attribute represents the scheduling policy to be assigned to the new
process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER is set in the spawn-
flags attribute). The default value of this attribute is unspecified.

RETURN VALUE
Upon successful completion, posix_spawnattr_getschedpolicy() shall return zero and store the
value of the spawn-schedpolicy attribute of attr into the object referenced by the schedpolicy
parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, posix_spawnattr_setschedpolicy() shall return zero; otherwise, an
error number shall be returned to indicate the error.

ERRORS
These functions may fail if:

[EINVAL] The value specified by attr is invalid.

The posix_spawnattr_setschedpolicy() function may fail if:

[EINVAL] The value of the attribute being set is not valid.

EXAMPLES
None.

APPLICATION USAGE
These functions are part of the Spawn and Process Scheduling options and need not be provided
on all implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
posix_spawn(), posix_spawnattr_destroy(), posix_spawnattr_init(), posix_spawnattr_getsigdefault(),
posix_spawnattr_getflags(), posix_spawnattr_getpgroup(), posix_spawnattr_getschedparam(),
posix_spawnattr_getsigmask(), posix_spawnattr_setsigdefault(), posix_spawnattr_setflags(),
posix_spawnattr_setpgroup(), posix_spawnattr_setschedparam(), posix_spawnattr_setsigmask(),
posix_spawnattr_getschedpolicy()

The function `posix_spawnattr_getschedpolicy()` is used to retrieve the scheduling policy associated with a `spawnattr_t` resource described structure. This function is part of the System Interfaces library, as indicated by the `<sched.h>` and `<spawn.h>` headers.

**CHANGE HISTORY**
NAME
posix_spawnattr_getsigdefault, posix_spawnattr_setsigdefault — get and set the spawn-
sigdefault attribute of a spawn attributes object (ADVANCED REALTIME)

SYNOPSIS
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigdefault(const posix_spawnattr_t *
    restrict attr, sigset_t *restrict sigdefault);
int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigdefault);

DESCRIPTION
The posix_spawnattr_getsigdefault() function shall obtain the value of the spawn-sigdefault
attribute from the attributes object referenced by attr.

The posix_spawnattr_setsigdefault() function shall set the spawn-sigdefault attribute in an
initialized attributes object referenced by attr.

The spawn-sigdefault attribute represents the set of signals to be forced to default signal handling
in the new process image (if POSIX_SPAWN_SETSIGDEF is set in the spawn-flags attribute) by a
spawn operation. The default value of this attribute shall be an empty signal set.

RETURN VALUE
Upon successful completion, posix_spawnattr_getsigdefault() shall return zero and store the value
of the spawn-sigdefault attribute of attr into the object referenced by the sigdefault parameter;
otherwise, an error number shall be returned to indicate the error.

Upon successful completion, posix_spawnattr_setsigdefault() shall return zero; otherwise, an error
number shall be returned to indicate the error.

ERRORS
These functions may fail if:

[EINVAL] The value specified by attr is invalid.

The posix_spawnattr_setsigdefault() function may fail if:

[EINVAL] The value of the attribute being set is not valid.

EXAMPLES
None.

APPLICATION USAGE
These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
posix_spawn(), posix_spawnattr_destroy(), posix_spawnattr_init(), posix_spawnattr_getflags(),
posix_spawnattr_getpgroup(), posix_spawnattr_getschedparam(), posix_spawnattr_getschedpolicy(),
posix_spawnattr_getsigmask(), posix_spawnattr_setflags(), posix_spawnattr_setpgroup(),
posix_spawnattr_setschedparam(), posix_spawnattr_setschedpolicy(), posix_spawnattr_setsigmask(),
posix_spawnp(), the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>, <spawn.h>
CHANG HISTORY
NAME
posix_spawnattr_getsigmask, posix_spawnattr_setsigmask — get and set the spawn-sigmask attribute of a spawn attributes object (ADVANCED REALTIME)

SYNOPSIS
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict attr, sigset_t *restrict sigmask);
int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr, const sigset_t *restrict sigmask);

DESCRIPTION
The posix_spawnattr_getsigmask() function shall obtain the value of the spawn-sigmask attribute from the attributes object referenced by attr.

The posix_spawnattr_setsigmask() function shall set the spawn-sigmask attribute in an initialized attributes object referenced by attr.

The spawn-sigmask attribute represents the signal mask in effect in the new process image of a spawn operation (if POSIX_SPAWN_SETSIGMASK is set in the spawn-flags attribute). The default value of this attribute is unspecified.

RETURN VALUE
Upon successful completion, posix_spawnattr_getsigmask() shall return zero and store the value of the spawn-sigmask attribute of attr into the object referenced by the sigmask parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, posix_spawnattr_setsigmask() shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS
These functions may fail if:

[EINVAL] The value specified by attr is invalid.

The posix_spawnattr_setsigmask() function may fail if:

[EINVAL] The value of the attribute being set is not valid.

EXAMPLES
None.

APPLICATION USAGE
These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
posix_spawn(), posix_spawnattr_destroy(), posix_spawnattr_init(), posix_spawnattr_getsigdefault(), posix_spawnattr_getflags(), posix_spawnattr_getpgroup(), posix_spawnattr_getschedparam(), posix_spawnattr_getschedpolicy(), posix_spawnattr_setsigdefault(), posix_spawnattr_setflags(), posix_spawnattr_setpgroup(), posix_spawnattr_setschedparam(), posix_spawnattr_setschedpolicy(), posix_spawnp(), the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>, <spawn.h>
posix_spawnattr_getsigmask()

CHANGE HISTORY
NAME
posix_spawnattr_init — initialize the spawn attributes object (ADVANCED REALTIME)

SYNOPSIS
#include <spawn.h>

int posix_spawnattr_init(posix_spawnattr_t *attr);

DESCRIPTION
Refer to posix_spawnattr_destroy().
NAME
posix_spawnattr_setflags — set the spawn-flags attribute of a spawn attributes object
(ADVANCED REALTIME)

SYNOPSIS
#include <spawn.h>
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);

DESCRIPTION
Refer to posix_spawnattr_getflags().
NAME
posix_spawnattr_setpgroup — set the spawn-pgroup attribute of a spawn attributes object
(ADVANCED REALTIME)

SYNOPSIS
#include <spawn.h>

int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);

DESCRIPTION
Refer to posix_spawnattr_getpgroup().
NAME
posix_spawnattr_setschedparam — set the spawn-schedparam attribute of a spawn attributes object (ADVANCED REALTIME)

SYNOPSIS
#include <sched.h>
#include <spawn.h>

int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
       const struct sched_param *restrict schedparam);

DESCRIPTION
Refer to posix_spawnattr_getschedparam().
posix_spawnattr_setschedpolicy()

NAME
posix_spawnattr_setschedpolicy — set the spawn-schedpolicy attribute of a spawn attributes object (ADVANCED REALTIME)

SYNOPSIS
#include <sched.h>
#include <spawn.h>

int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr, int schedpolicy);

DESCRIPTION
Refer to posix_spawnattr_getschedpolicy().
NAME
posix_spawnattr_setsigdefault — set the spawn-sigdefault attribute of a spawn attributes object
(ADVANCED REALTIME)

SYNOPSIS
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
const sigset_t *restrict sigdefault);

DESCRIPTION
Refer to posix_spawnattr_getsigdefault().
NAME
posix_spawnattr_setsigmask — set the spawn-sigmask attribute of a spawn attributes object
(ADVANCED REALTIME)

SYNOPSIS
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
const sigset_t *restrict sigmask);

DESCRIPTION
Refer to posix_spawnattr_getsigmask().
NAME
posix_spawnp — spawn a process (ADVANCED REALTIME)

SYNOPSIS
#include <spawn.h>

int posix_spawnp(pid_t *restrict pid, const char *restrict file,
                 const posix_spawn_file_actions_t *file_actions,
                 const posix_spawnattr_t *restrict attrp,
                 char *const argv[restrict], char *const envp[restrict]);

DESCRIPTION
Refer to posix_spawn().
NAME

posix_trace_attr_destroy, posix_trace_attr_init — destroy and initialize the trace stream attributes object (TRACING)

SYNOPSIS

```
#include <trace.h>

int posix_trace_attr_destroy(trace_attr_t *attr);
int posix_trace_attr_init(trace_attr_t *attr);
```

DESCRIPTION

The `posix_trace_attr_destroy()` function shall destroy an initialized trace attributes object. A destroyed `attr` attributes object can be reinitialized using `posix_trace_attr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

The `posix_trace_attr_init()` function shall initialize a trace attributes object `attr` with the default value for all of the individual attributes used by a given implementation. The read-only `generation-version` and `clock-resolution` attributes of the newly initialized trace attributes object shall be set to their appropriate values (see Section 2.11.1.2 (on page 75)).

Results are undefined if `posix_trace_attr_init()` is called specifying an already initialized `attr` attributes object.

Implementations may add extensions to the trace attributes object structure as permitted in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 2, Conformance.

The resulting attributes object (possibly modified by setting individual attributes values), when used by `posix_trace_create()`, defines the attributes of the trace stream created. A single attributes object can be used in multiple calls to `posix_trace_create()`. After one or more trace streams have been created using an attributes object, any function affecting that attributes object, including destruction, shall not affect any trace stream previously created. An initialized attributes object also serves to receive the attributes of an existing trace stream or trace log when calling the `posix_trace_get_attr()` function.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

ERRORS

The `posix_trace_attr_destroy()` function may fail if:

- [EINVAL] The value of `attr` is invalid.

The `posix_trace_attr_init()` function shall fail if:

- [ENOMEM] Insufficient memory exists to initialize the trace attributes object.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.
FUTURE DIRECTIONS
None.

SEE ALSO
posix_trace_create(), posix_trace_get_attr(), uname(), the Base Definitions volume of IEEE Std 1003.1-2001, <trace.h>

CHANGE HISTORY
IEEE PASC Interpretation 1003.1 #123 is applied.
NAME
posix_trace_attr_getclockres, posix_trace_attr_getcreatetime, posix_trace_attr_getgenversion,
posix_trace_attr_getname, posix_trace_attr_setname — retrieve and set information about a
trace stream (TRACING)

SYNOPSIS
TRC
#include <time.h>
#include <trace.h>

int posix_trace_attr_getclockres(const trace_attr_t *attr,
       struct timespec *resolution);
int posix_trace_attr_getcreatetime(const trace_attr_t *attr,
       struct timespec *createtime);

#include <trace.h>

int posix_trace_attr_getgenversion(const trace_attr_t *attr,
       char *genversion);
int posix_trace_attr_getname(const trace_attr_t *attr,
       char *tracename);
int posix_trace_attr_setname(trace_attr_t *attr,
       const char *tracename);

DESCRIPTION
The posix_trace_attr_getclockres() function shall copy the clock resolution of the clock used to
generate timestamps from the clock-resolution attribute of the attributes object pointed to by the
attr argument into the structure pointed to by the resolution argument.

The posix_trace_attr_getcreatetime() function shall copy the trace stream creation time from the
creation-time attribute of the attributes object pointed to by the attr argument into the structure
pointed to by the createtime argument. The creation-time attribute shall represent the time of
creation of the trace stream.

The posix_trace_attr_getgenversion() function shall copy the string containing version information
from the generation-version attribute of the attributes object pointed to by the attr argument into
the string pointed to by the genversion argument. The genversion argument shall be the address of
a character array which can store at least {TRACE_NAME_MAX} characters.

The posix_trace_attr_getname() function shall copy the string containing the trace name from the
trace-name attribute of the attributes object pointed to by the attr argument into the string
pointed to by the tracename argument. The tracename argument shall be the address of a character
array which can store at least {TRACE_NAME_MAX} characters.

The posix_trace_attr_setname() function shall set the name in the trace-name attribute of the
attributes object pointed to by the attr argument, using the trace name string supplied by the
tracename argument. If the supplied string contains more than {TRACE_NAME_MAX}
characters, the name copied into the trace-name attribute may be truncated to one less than the
length of {TRACE_NAME_MAX} characters. The default value is a null string.

RETURN VALUE
Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
return the corresponding error number.

If successful, the posix_trace_attr_getclockres() function stores the clock-resolution attribute value
in the object pointed to by resolution. Otherwise, the content of this object is unspecified.
If successful, the `posix_trace_attr_getcreatetime()` function stores the trace stream creation time in the object pointed to by `createtime`. Otherwise, the content of this object is unspecified.

If successful, the `posix_trace_attr_getgenversion()` function stores the trace version information in the string pointed to by `genversion`. Otherwise, the content of this string is unspecified.

If successful, the `posix_trace_attr_getname()` function stores the trace name in the string pointed to by `tracename`. Otherwise, the content of this string is unspecified.

**ERRORS**

The `posix_trace_attr_getclockres()`, `posix_trace_attr_getcreatetime()`, `posix_trace_attr_getgenversion()`, and `posix_trace_attr_getname()` functions may fail if:

[EINVAL] The value specified by one of the arguments is invalid.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`posix_trace_attr_init()`, `posix_trace_create()`, `posix_trace_get_attr()`, `uname()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<time.h>`, `<trace.h>`

**CHANGE HISTORY**

NAME
posix_trace_attr_getinherited, posix_trace_attr_getlogfullpolicy,
posix_trace_attr_getstreamfullpolicy, posix_trace_attr_setinherited,
posix_trace_attr_setlogfullpolicy, posix_trace_attr_setstreamfullpolicy — retrieve and set the
behavior of a trace stream (TRACING)

SYNOPSIS
#include <trace.h>

int posix_trace_attr_getinherited(const trace_attr_t *restrict attr,
       int *restrict inheritancepolicy);
int posix_trace_attr_getlogfullpolicy(const trace_attr_t *restrict attr,
       int *restrict logpolicy);
int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *attr,
       int *streampolicy);
int posix_trace_attr_setinherited(trace_attr_t *attr,
       int inheritancepolicy);
int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
       int logpolicy);
int posix_trace_attr_setstreamfullpolicy(trace_attr_t *attr,
       int streampolicy);

DESCRIPTION
The posix_trace_attr_getinherited() and posix_trace_attr_setinherited() functions, respectively, shall
get and set the inheritance policy stored in the inheritance attribute for traced processes across
the fork() and spawn() operations. The inheritance attribute of the attributes object pointed to by
the attr argument shall be set to one of the following values defined by manifest constants in the
#include <trace.h> header:

POSIX_TRACE_CLOSE_FOR_CHILD
After a fork() or spawn() operation, the child shall not be traced, and tracing of the parent
shall continue.

POSIX_TRACE_INHERITED
After a fork() or spawn() operation, if the parent is being traced, its child shall be
concurrently traced using the same trace stream.

The default value for the inheritance attribute is POSIX_TRACE_CLOSE_FOR_CHILD.

The posix_trace_attr_getlogfullpolicy() and posix_trace_attr_setlogfullpolicy() functions,
respectively, shall get and set the trace log full policy stored in the log-full-policy attribute of the
attributes object pointed to by the attr argument.

The log-full-policy attribute shall be set to one of the following values defined by manifest
constants in the <trace.h> header:

POSIX_TRACE_LOOP
The trace log shall loop until the associated trace stream is stopped. This policy means that
when the trace log gets full, the file system shall reuse the resources allocated to the oldest
trace events that were recorded. In this way, the trace log will always contain the most
recent trace events flushed.

POSIX_TRACE_UNTIL_FULL
The trace stream shall be flushed to the trace log until the trace log is full. This condition can
be deduced from the posix_log_full_status member status (see the posix_trace_status_info
structure defined in <trace.h>). The last recorded trace event shall be the
POSIX_TRACE_STOP trace event.

POSIX_TRACE_APPEND

The associated trace stream shall be flushed to the trace log without log size limitation. If the application specifies POSIX_TRACE_APPEND, the implementation shall ignore the log-max-size attribute.

The default value for the log-full-policy attribute is POSIX_TRACE_LOOP.

The posix_trace_attr_getstreamfullpolicy() and posix_trace_attr_setstreamfullpolicy() functions, respectively, shall get and set the trace stream full policy stored in the stream-full-policy attribute of the attributes object pointed to by the attr argument.

The stream-full-policy attribute shall be set to one of the following values defined by manifest constants in the <trace.h> header:

POSIX_TRACE_LOOP

The trace stream shall loop until explicitly stopped by the posix_trace_stop() function. This policy means that when the trace stream is full, the trace system shall reuse the resources allocated to the oldest trace events recorded. In this way, the trace stream will always contain the most recent trace events recorded.

POSIX_TRACE_UNTIL_FULL

The trace stream will run until the trace stream resources are exhausted. Then the trace stream will stop. This condition can be deduced from posix_stream_status and posix_stream_full_status (see the posix_trace_status_info structure defined in <trace.h>). When this trace stream is read, a POSIX_TRACE_STOP trace event shall be reported after reporting the last recorded trace event. The trace system shall reuse the resources allocated to any trace events already reported—see the posix_trace_getnext_event(), posix_trace_trygetnext_event(), and posix_trace_timedgetnext_event() functions—or already flushed for an active trace stream with log if the Trace Log option is supported; see the posix_trace_flush() function. The trace system shall restart the trace stream when it is empty and may restart it sooner. A POSIX_TRACE_START trace event shall be reported before reporting the next recorded trace event.

POSIX_TRACE_FLUSH

If the Trace Log option is supported, this policy is identical to the POSIX_TRACE_UNTIL_FULL trace stream full policy except that the trace stream shall be flushed regularly as if posix_trace_flush() had been explicitly called. Defining this policy for an active trace stream without log shall be invalid.

The default value for the stream-full-policy attribute shall be POSIX_TRACE_LOOP for an active trace stream without log.

If the Trace Log option is supported, the default value for the stream-full-policy attribute shall be POSIX_TRACE_FLUSH for an active trace stream with log.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

If successful, the posix_trace_attr_getinherited() function shall store the inheritance attribute value in the object pointed to by inheritancepolicy. Otherwise, the content of this object is undefined.

If successful, the posix_trace_attr_getlogfullpolicy() function shall store the log-full-policy attribute value in the object pointed to by logpolicy. Otherwise, the content of this object is undefined.

If successful, the posix_trace_attr_getstreamfullpolicy() function shall store the stream-full-policy attribute value in the object pointed to by streampolicy. Otherwise, the content of this object is undefined.
undefined.

ERRORS
These functions may fail if:

[EINVAL] The value specified by at least one of the arguments is invalid.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fork(), posix_trace_attr_init(), posix_trace_create(), posix_trace_flush(), posix_trace_get_attr(),
posix_trace_getnext_event(), posix_trace_start(), posix_trace_timedgetnext_event(), the Base
Definitions volume of IEEE Std 1003.1-2001, <trace.h>

CHANGE HISTORY
NAME
posix_trace_attr_getlogsize(), posix_trace_attr_getmaxdatasize,
posix_trace_attr_getmaxsystemeventsize, posix_trace_attr_getmaxusereventsize,
posix_trace_attr_getstreamsize, posix_trace_attr_setlogsize, posix_trace_attr_setmaxdatasize,
posix_trace_attr_setstreamsize — retrieve and set trace stream size attributes (TRACING)

SYNOPSIS
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_getlogsize(const trace_attr_t *restrict attr,
    size_t *restrict logsize);
int posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict attr,
    size_t *restrict maxdatasize);
int posix_trace_attr_getmaxsystemeventsize(
    const trace_attr_t *restrict attr,
    size_t *restrict eventsize);
int posix_trace_attr_getmaxusereventsize(
    const trace_attr_t *restrict attr,
    size_t data_len, size_t *restrict eventsize);
int posix_trace_attr_getstreamsize(const trace_attr_t *restrict attr,
    size_t *restrict streamsize);
int posix_trace_attr_setlogsize(trace_attr_t *attr,
    size_t logsize);
int posix_trace_attr_setmaxdatasize(trace_attr_t *attr,
    size_t maxdatasize);
int posix_trace_attr_setstreamsize(trace_attr_t *attr,
    size_t streamsize);

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.

The posix_trace_attr_setlogsize() function shall set the maximum allowed size, in bytes, in the log-max-size attribute of the attributes object pointed to by the attr argument, using the size value supplied by the logsize argument.

The trace log size shall be used if the log-full-policy attribute is set to POSIX_TRACE_LOOP or POSIX_TRACE_UNTIL_FULL. If the log-full-policy attribute is set to POSIX_TRACE_APPEND, the implementation shall ignore the log-max-size attribute.

The posix_trace_attr_getmaxdatasize() function shall copy the maximum user trace event data size, in bytes, from the max-data-size attribute of the attributes object pointed to by the attr argument into the variable pointed to by the maxdatasize argument. The default value for the max-data-size attribute is implementation-defined.

The posix_trace_attr_getmaxsystemeventsize() function shall calculate the maximum memory size, in bytes, required to store a single system trace event. This value is calculated for the trace stream attributes object pointed to by the attr argument and is returned in the variable pointed to by the eventsize argument.
The values returned as the maximum memory sizes of the user and system trace events shall be such that if the sum of the maximum memory sizes of a set of the trace events that may be recorded in a trace stream is less than or equal to the \textit{stream-min-size} attribute of that trace stream, the system provides the necessary resources for recording all those trace events, without loss.

The \texttt{posix_trace_attr_getmaxusereventsize()} function shall calculate the maximum memory size, in bytes, required to store a single user trace event generated by a call to \texttt{posix_trace_event()} with a \textit{data_len} parameter equal to the \textit{data_len} value specified in this call. This value is calculated for the trace stream attributes object pointed to by the \textit{attr} argument and is returned in the variable pointed to by the \textit{eventsize} argument.

The \texttt{posix_trace_attr_getstreamsize()} function shall copy the stream size, in bytes, from the \textit{stream-min-size} attribute of the attributes object pointed to by the \textit{attr} argument into the variable pointed to by the \textit{streamsize} argument.

This stream size is the current total memory size reserved for system and user trace events in the trace stream. The default value for the \textit{stream-min-size} attribute is implementation-defined. The stream size refers to memory used to store trace event records. Other stream data (for example, trace attribute values) shall not be included in this size.

The \texttt{posix_trace_attr_setmaxdatasize()} function shall set the maximum allowed size, in bytes, in the \textit{max-data-size} attribute of the attributes object pointed to by the \textit{attr} argument, using the size value supplied by the \textit{maxdatasize} argument. This maximum size is the maximum allowed size for the user data argument which may be passed to \texttt{posix_trace_event()}. The implementation shall be allowed to truncate data passed to \texttt{trace_user_event()} which is longer than \textit{maxdatasize}.

The \texttt{posix_trace_attr_setstreamsize()} function shall set the minimum allowed size, in bytes, in the \textit{stream-min-size} attribute of the attributes object pointed to by the \textit{attr} argument, using the size value supplied by the \textit{streamsize} argument.

\section*{RETURN VALUE}
Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

\section*{ERRORS}
These functions may fail if:

\begin{itemize}
\item \texttt{[EINVAL]} The value specified by one of the arguments is invalid.
\end{itemize}
posix_trace_attr_getlogsize()  System Interfaces

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
posix_trace_attr_init(), posix_trace_create(), posix_trace_event(), posix_trace_get_attr(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <trace.h>

CHANGE HISTORY
NAME
posix_trace_attr_getname — retrieve and set information about a trace stream (TRACING)

SYNOPSIS

```
#include <trace.h>

int posix_trace_attr_getname(const trace_attr_t *attr, char *tracename);
```

DESCRIPTION

Refer to `posix_trace_attr_getclockres()`. 
NAME
posix_trace_attr_getstreamfullpolicy — retrieve and set the behavior of a trace stream
(TRACING)

SYNOPSIS
#include <trace.h>

int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *attr,
int *streampolicy);

DESCRIPTION
Refer to posix_trace_attr_getinherited().
NAME
posix_trace_attr_getstreamsize — retrieve and set trace stream size attributes (TRACING)

SYNOPSIS
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_getstreamsize(const trace_attr_t *restrict attr,
    size_t *restrict streamsize);
```

DESCRIPTION
Refer to `posix_trace_attr_getlogsize()`.
NAME
posix_trace_attr_init — initialize the trace stream attributes object (TRACING)

SYNOPSIS
#include <trace.h>

int posix_trace_attr_init(trace_attr_t *attr);

DESCRIPTION
Refer to posix_trace_attr_destroy().
NAME
posix_trace_attr_setinherited, posix_trace_attr_setlogfullpolicy — retrieve and set the behavior of a trace stream (TRACING)

SYNOPSIS
#include <trace.h>

int posix_trace_attr_setinherited(trace_attr_t *attr,
  int inheritancepolicy);
int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
  int logpolicy);

DESCRIPTION
Refer to posix_trace_attr_getinherited().
posix_trace_attr_setlogsize()  System Interfaces

NAME
posix_trace_attr_setlogsize, posix_trace_attr_setmaxdatasize — retrieve and set trace stream size attributes (TRACING)

SYNOPSIS
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_setlogsize(trace_attr_t *attr, size_t logsize);
int posix_trace_attr_setmaxdatasize(trace_attr_t *attr, size_t maxdatasize);
```

DESCRIPTION
Refer to `posix_trace_attr_getlogsize()`.
NAME
posix_trace_attr_setname — retrieve and set information about a trace stream (TRACING)

SYNOPSIS

```c
#include <trace.h>

int posix_trace_attr_setname(trace_attr_t *attr,
const char *tracename);
```

DESCRIPTION

Refer to `posix_trace_attr_getclockres()`.
NAME
posix_trace_attr_setstreamfullpolicy — retrieve and set the behavior of a trace stream (TRACING)

SYNOPSIS
```
#include <trace.h>

int posix_trace_attr_setstreamfullpolicy(trace_attr_t *attr,
                                      int logpolicy);
```

DESCRIPTION
Refer to `posix_trace_attr_getinherited()`.
posix_trace_attr_setstreamsize() — retrieve and set trace stream size attributes (TRACING)

#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_setstreamsize(trace_attr_t *attr,
        size_t streamsize);

Refer to posix_trace_attr_getlogsize().
posix_trace_clear()

NAME
posix_trace_clear — clear trace stream and trace log (TRACING)

SYNOPSIS

```c
#include <sys/types.h>
#include <trace.h>

int posix_trace_clear(trace_id_t trid);
```

DESCRIPTION
The `posix_trace_clear()` function shall reinitialize the trace stream identified by the argument `trid` as if it were returning from the `posix_trace_create()` function, except that the same allocated resources shall be reused, the mapping of trace event type identifiers to trace event names shall be unchanged, and the trace stream status shall remain unchanged (that is, if it was running, it remains running and if it was suspended, it remains suspended).

All trace events in the trace stream recorded before the call to `posix_trace_clear()` shall be lost.

The `posix_stream_full_status` status shall be set to POSIX_TRACE_NOT_FULL. There is no guarantee that all trace events that occurred during the `posix_trace_clear()` call are recorded; the behavior with respect to trace points that may occur during this call is unspecified.

If the Trace Log option is supported and the trace stream has been created with a log, the `posix_trace_clear()` function shall reinitialize the trace stream with the same behavior as if the trace stream was created without the log, plus it shall reinitialize the trace log associated with the trace stream identified by the argument `trid` as if it were returning from the `posix_trace_create_withlog()` function, except that the same allocated resources, for the trace log, may be reused and the associated trace stream status remains unchanged. The first trace event recorded in the trace log after the call to `posix_trace_clear()` shall be the same as the first trace event recorded in the active trace stream after the call to `posix_trace_clear()`. The `posix_log_full_status` status shall be set to POSIX_TRACE_NOT_FULL. There is no guarantee that all trace events that occurred during the `posix_trace_clear()` call are recorded in the trace log; the behavior with respect to trace points that may occur during this call is unspecified. If the log full policy is POSIX_TRACE_APPEND, the effect of a call to this function is unspecified for the trace log associated with the trace stream identified by the `trid` argument.

RETURN VALUE
Upon successful completion, the `posix_trace_clear()` function shall return a value of zero.

Otherwise, it shall return the corresponding error number.

ERRORS
The `posix_trace_clear()` function shall fail if:

[EINVAL] The value of the `trid` argument does not correspond to an active trace stream.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO

posix_trace_attr_init(), posix_trace_create(), posix_trace_flush(), posix_trace_get_attr(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <trace.h>

CHANGE HISTORY


IEEE PASC Interpretation 1003.1 #123 is applied.
NAME
 posix_trace_close, posix_trace_open, posix_trace_rewind — trace log management (TRACING)

SYNOPSIS
#include <trace.h>

int posix_trace_close(trace_id_t trid);
int posix_trace_open(int file_desc, trace_id_t *trid);
int posix_trace_rewind(trace_id_t trid);

DESCRIPTION
The posix_trace_close() function shall deallocate the trace log identifier indicated by trid, and all
of its associated resources. If there is no valid trace log pointed to by the trid, this function shall
fail.

The posix_trace_open() function shall allocate the necessary resources and establish the
connection between a trace log identified by the file_desc argument and a trace stream identifier
identified by the object pointed to by the trid argument. The file_desc argument should be a valid
open file descriptor that corresponds to a trace log. The file_desc argument shall be open for
reading. The current trace event timestamp, which specifies the timestamp of the trace event
that will be read by the next call to posix_trace_getnext_event(), shall be set to the timestamp of
the oldest trace event recorded in the trace log identified by trid.

The posix_trace_open() function shall return a trace stream identifier in the variable pointed to by
the trid argument, that may only be used by the following functions:

posix_trace_close()     posix_trace_get_attr()
posix_trace_eventid_equal()     posix_trace_get_status()
posix_trace_eventid_get_name()     posix_trace_getnext_event()
posix_trace_eventtypelist_getnext_id()     posix_trace_rewind()
posix_trace_eventtypelist_rewind()

In particular, notice that the operations normally used by a trace controller process, such as
posix_trace_start(), posix_trace_stop(), or posix_trace_shutdown(), cannot be invoked using the
trace stream identifier returned by the posix_trace_open() function.

The posix_trace_rewind() function shall reset the current trace event timestamp, which specifies
the timestamp of the trace event that will be read by the next call to posix_trace_getnext_event(),
to the timestamp of the oldest trace event recorded in the trace log identified by trid.

RETURN VALUE
Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
return the corresponding error number.

If successful, the posix_trace_open() function stores the trace stream identifier value in the object
pointed to by trid.

ERRORS
The posix_trace_open() function shall fail if:

[EINTR] The operation was interrupted by a signal and thus no trace log was opened.
[EINVAL] The object pointed to by file_desc does not correspond to a valid trace log.

The posix_trace_close() and posix_trace_rewind() functions may fail if:

[EINVAL] The object pointed to by trid does not correspond to a valid trace log.
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
posix_trace_get_attr(), posix_trace_get_filter(), posix_trace_getnext_event(), the Base Definitions volume of IEEE Std 1003.1-2001, <trace.h>

CHANGE HISTORY
IEEE PASC Interpretation 1003.1 #123 is applied.
NAME

posix_trace_create, posix_trace_create_withlog, posix_trace_flush, posix_trace_shutdown —
trace stream initialization, flush, and shutdown from a process (TRACING)

SYNOPSIS

#include <sys/types.h>
#include <trace.h>

int posix_trace_create(pid_t pid,
    const trace_attr_t *restrict attr,
    trace_id_t *restrict trid);

int posix_trace_create_withlog(pid_t pid,
    const trace_attr_t *restrict attr, int file_desc,
    trace_id_t *restrict trid);

int posix_trace_flush(trace_id_t trid);

int posix_trace_shutdown(trace_id_t trid);

DESCRIPTION

The posix_trace_create() function shall create an active trace stream. It allocates all the resources
needed by the trace stream being created for tracing the process specified by pid in accordance
with the attr argument. The attr argument represents the initial attributes of the trace stream and
shall have been initialized by the function posix_trace_attr_init() prior to the posix_trace_create()
call. If the argument attr is NULL, the default attributes shall be used. The attr attributes object
shall be manipulated through a set of functions described in the posix_trace_attr family of
functions. If the attributes of the object pointed to by attr are modified later, the attributes of the
trace stream shall not be affected. The creation-time attribute of the newly created trace stream
shall be set to the value of the system clock, if the Timers option is not supported, or to the value
of the CLOCK_REALTIME clock, if the Timers option is supported.

The pid argument represents the target process to be traced. If the process executing this
function does not have appropriate privileges to trace the process identified by pid, an error shall
be returned. If the pid argument is zero, the calling process shall be traced.

The posix_trace_create() function shall store the trace stream identifier of the new trace stream in
the object pointed to by the trid argument. This trace stream identifier shall be used in
subsequent calls to control tracing. The trid argument may only be used by the following
functions:

posix_trace_clear()  posix_trace_getnext_event()
posix_trace_eventid_equal()  posix_trace_shutdown()
posix_trace_eventid_get_name()  posix_trace_start()
posix_trace_eventtypelist_getnext_id()  posix_trace_stop()
posix_trace_eventtypelist_rewind()  posix_trace_timedgetnext_event()
posix_trace_get_attr()  posix_trace_trid_eventid_open()
posix_trace_get_status()  posix_trace_trygetnext_event()

If the Trace Event Filter option is supported, the following additional functions may use the trid
argument:

posix_trace_get_filter()  posix_trace_set_filter()
In particular, notice that the operations normally used by a trace analyzer process, such as
posix_trace_rewind() or posix_trace_close(), cannot be invoked using the trace stream identifier
returned by the posix_trace_create() function.

A trace stream shall be created in a suspended state. If the Trace Event Filter option is
supported, its trace event type filter shall be empty.

The posix_trace_create() function may be called multiple times from the same or different
processes, with the system-wide limit indicated by the runtime invariant value
/TRACE_SYS_MAX[, which has the minimum value /_POSIX_TRACE_SYS_MAX).

The trace stream identifier returned by the posix_trace_create() function in the argument pointed
to by trid is valid only in the process that made the function call. If it is used from another
process, that is a child process, in functions defined in IEEE Std 1003.1-2001, these functions shall
return with the error [EINVAL].

The posix_trace_create_withlog() function shall be equivalent to posix_trace_create(), except that it
associates a trace log with this stream. The file_desc argument shall be the file descriptor
designating the trace log destination. The function shall fail if this file descriptor refers to a file
with a file type that is not compatible with the log policy associated with the trace log. The list of
the appropriate file types that are compatible with each log policy is implementation-defined.

The posix_trace_create_withlog() function shall return in the parameter pointed to by trid the trace
stream identifier, which uniquely identifies the newly created trace stream, and shall be used in
subsequent calls to control tracing. The trid argument may only be used by the following
functions:

posix_trace_get_filter()  posix_trace_set_filter()
posix_trace_getnext_event()  posix_trace_shutdown()
posix_trace_eventid_equal()  posix_trace_start()
posix_trace_eventid_get_name()  posix_trace_stop()
posix_trace_eventtypelist_getnext_id()  posix_trace_timedgetnext_event()
posix_trace_eventtypelist_rewind()  posix_trace_trid_eventid_open()
posix_trace_flush()  posix_trace_trygetnext_event()
posix_trace_get_attr()  posix_trace_get_status()
posix_trace_get_status()
If flushing the trace stream causes the resulting trace log to become full, the trace log full policy shall be applied. If the trace log-full-policy attribute is set, the following occurs:

- **POSIX_TRACE_UNTIL_FULL**: The trace events that have not yet been flushed shall be discarded.
- **POSIX_TRACE_LOOP**: The trace events that have not yet been flushed shall be written to the beginning of the trace log, overwriting previous trace events stored there.
- **POSIX_TRACE_APPEND**: The trace events that have not yet been flushed shall be appended to the trace log.

The `posix_trace_shutdown()` function shall stop the tracing of trace events in the trace stream identified by `trid`, as if `posix_trace_stop()` had been invoked. The `posix_trace_shutdown()` function shall free all the resources associated with the trace stream.

The `posix_trace_shutdown()` function shall not return until all the resources associated with the trace stream have been freed. When the `posix_trace_shutdown()` function returns, the `trid` argument becomes an invalid trace stream identifier. A call to this function shall unconditionally deallocate the resources regardless of whether all trace events have been retrieved by the analyzer process. Any thread blocked on one of the `trace_getnext_event()` functions (which specified this `trid`) before this call is unblocked with the error [EINVAL].

If the process exits, invokes a member of the exec family of functions, or is terminated, the trace streams that the process had created and that have not yet been shut down, shall be automatically shut down as if an explicit call were made to the `posix_trace_shutdown()` function.

For an active trace stream with log, when the `posix_trace_shutdown()` function is called, all trace events that have not yet been flushed to the trace log shall be flushed, as in the `posix_trace_flush()` function, and the trace log shall be closed.

When a trace log is closed, all the information that may be retrieved later from the trace log through the trace interface shall have been written to the trace log. This information includes the trace attributes, the list of trace event types (with the mapping between trace event names and trace event type identifiers), and the trace status.

In addition, unspecified information shall be written to the trace log to allow detection of a valid trace log during the `posix_trace_open()` operation.

The `posix_trace_shutdown()` function shall not return until all trace events have been flushed.

**RETURN VALUE**

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

The `posix_trace_create()` and `posix_trace_create_withlog()` functions store the trace stream identifier value in the object pointed to by `trid`, if successful.

**ERRORS**

The `posix_trace_create()` and `posix_trace_create_withlog()` functions shall fail if:

- `[EAGAIN]` No more trace streams can be started now. [TRACE_SYS_MAX] has been exceeded.
- `[EINTR]` The operation was interrupted by a signal. No trace stream was created.
- `[EINVAL]` One or more of the trace parameters specified by the `attr` parameter is invalid.
The implementation does not currently have sufficient memory to create the trace stream with the specified parameters.

The caller does not have appropriate privilege to trace the process specified by \texttt{pid}.

The \texttt{pid} argument does not refer to an existing process.

The \texttt{posix_trace_create_withlog()} function shall fail if:

- \textbf{[ENOMEM]} The implementation does not currently have sufficient memory to create the trace stream with the specified parameters.
- \textbf{[EPERM]} The caller does not have appropriate privilege to trace the process specified by \texttt{pid}.
- \textbf{[ESRCH]} The \texttt{pid} argument does not refer to an existing process.

The \texttt{posix_trace_flush()} and \texttt{posix_trace_shutdown()} functions shall fail if:

- \textbf{[EINVAL]} The value of the \texttt{trid} argument does not correspond to an active trace stream with log.
- \textbf{[EFBIG]} The trace log file has attempted to exceed an implementation-defined maximum file size.
- \textbf{[ENOSPC]} No space left on device.

The \texttt{posix_trace_flush()} and \texttt{posix_trace_shutdown()} functions shall fail if:

- \textbf{[EINVAL]} The value of the \texttt{trid} argument does not correspond to an active trace stream with log.
- \textbf{[EFBIG]} The trace log file has attempted to exceed an implementation-defined maximum file size.
- \textbf{[ENOSPC]} No space left on device.

\textbf{EXAMPLES}

None.

\textbf{APPLICATION USAGE}

None.

\textbf{RATIONALE}

None.

\textbf{FUTURE DIRECTIONS}

None.

\textbf{SEE ALSO}

- \texttt{clock_getres()}, \texttt{exec}, \texttt{posix_trace_attr_init()}, \texttt{posix_trace_close()}, \texttt{posix_trace_create()},
- \texttt{posix_trace_eventid_equal()}, \texttt{posix_trace_eventtypelist_getnext_id()}, \texttt{posix_trace_flush()},
- \texttt{posix_trace_get_attr()}, \texttt{posix_trace_get_filter()}, \texttt{posix_trace_get_status()}, \texttt{posix_trace_getnext_event()},
- \texttt{posix_trace_open()}, \texttt{posix_trace_set_filter()}, \texttt{posix_trace_shutdown()}, \texttt{posix_trace_start()},
- \texttt{posix_trace_timedgetnext_event()}, \texttt{posix_trace_trid_eventid_open()}, \texttt{posix_trace_start()}, \texttt{time()}, \texttt{the Base Definitions volume of IEEE Std 1003.1-2001}, \texttt{<sys/types.h>}, \texttt{<trace.h>}

\textbf{CHANGE HISTORY}

NAME
posix_trace_event, posix_trace_eventid_open — trace functions for instrumenting application code (TRACING)

SYNOPSIS

```c
#include <sys/types.h>
#include <trace.h>

void posix_trace_event(trace_event_id_t event_id,
    const void *restrict data_ptr, size_t data_len);

int posix_trace_eventid_open(const char *restrict event_name,
    trace_event_id_t *restrict event_id);
```

DESCRIPTION

The `posix_trace_event()` function shall record the `event_id` and the user data pointed to by `data_ptr` in the trace stream into which the calling process is being traced and in which `event_id` is not filtered out. If the total size of the user trace event data represented by `data_len` is not greater than the declared maximum size for user trace event data, then the `truncation-status` attribute of the trace event recorded is POSIX_TRACE_NOT_TRUNCATED. Otherwise, the user trace event data is truncated to this declared maximum size and the `truncation-status` attribute of the trace event recorded is POSIX_TRACE_TRUNCATED_RECORD.

If there is no trace stream created for the process or if the created trace stream is not running, or if the trace event specified by `event_id` is filtered out in the trace stream, the `posix_trace_event()` function shall have no effect.

The `posix_trace_eventid_open()` function shall associate a user trace event name with a trace event type identifier for the calling process. The trace event name is the string pointed to by the argument `event_name`. It shall have a maximum of {TRACE_EVENT_NAME_MAX} characters (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). The number of user trace event type identifiers that can be defined for any given process is limited by the maximum value {TRACE_USER_EVENT_MAX}, which has the minimum value {POSIX_TRACE_USER_EVENT_MAX}.

If the Trace Inherit option is not supported, the `posix_trace_eventid_open()` function shall associate the user trace event name pointed to by the `event_name` argument with a trace event type identifier that is unique for the traced process, and is returned in the variable pointed to by the `event_id` argument. If the user trace event name has already been mapped for the traced process, then the previously assigned trace event type identifier shall be returned. If the per-process user trace event name limit represented by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined POSIX_TRACE_UNNAMED_EVENT (Table 2-7 (on page 79)) user trace event shall be returned.

If the Trace Inherit option is supported, the `posix_trace_eventid_open()` function shall associate the user trace event name pointed to by the `event_name` argument with a trace event type identifier that is unique for all the processes being traced in this same trace stream, and is returned in the variable pointed to by the `event_id` argument. If the user trace event name has already been mapped for the traced processes, then the previously assigned trace event type identifier shall be returned. If the per-process user trace event name limit represented by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined POSIX_TRACE_UNNAMED_USEREVENT (Table 2-7 (on page 79)) user trace event shall be returned.

Note: The above procedure, together with the fact that multiple processes can only be traced into the same trace stream by inheritance, ensure that all the processes that are traced into a trace stream have the same mapping of trace event names to trace event type identifiers.
If there is no trace stream created, the `posix_trace_eventid_open()` function shall store this information for future trace streams created for this process.

**RETURN VALUE**

No return value is defined for the `posix_trace_event()` function.

Upon successful completion, the `posix_trace_eventid_open()` function shall return a value of zero. Otherwise, it shall return the corresponding error number. The `posix_trace_eventid_open()` function stores the trace event type identifier value in the object pointed to by `event_id`, if successful.

**ERRORS**

The `posix_trace_eventid_open()` function shall fail if:

- `[ENAMETOOLONG]`

  The size of the name pointed to by the `event_name` argument was longer than the implementation-defined value `{TRACE_EVENT_NAME_MAX}`.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

Table 2-7 (on page 79), `posix_trace_start()`, `posix_trace_trid_eventid_open()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/types.h>`, `<trace.h>`

**CHANGE HISTORY**


IEEE PASC Interpretation 1003.1 #123 is applied.

IEEE PASC Interpretation 1003.1 #127 is applied, correcting some editorial errors in the names of the `posix_trace_eventid_open()` function and the `event_id` argument.
NAME
posix_trace_eventid_equal, posix_trace_eventid_get_name, posix_trace_trid_eventid_open —
manipulate the trace event type identifier (TRACING)

SYNOPSIS
#include <trace.h>

int posix_trace_eventid_equal(trace_id_t trid, trace_event_id_t event1,
  trace_event_id_t event2);

int posix_trace_eventid_get_name(trace_id_t trid,
  trace_event_id_t event, char *event_name);

int posix_trace_trid_eventid_open(trace_id_t trid,
  const char *restrict event_name,
  trace_event_id_t *restrict event);

DESCRIPTION
The posix_trace_eventid_equal() function shall compare the trace event type identifiers event1 and
event2 from the same trace stream or the same trace log identified by the trid argument. If the
trace event type identifiers event1 and event2 are from different trace streams, the return value
shall be unspecified.

The posix_trace_eventid_get_name() function shall return, in the argument pointed to by
event_name, the trace event name associated with the trace event type identifier identified by the
argument event, for the trace stream or for the trace log identified by the trid argument. The
name of the trace event shall have a maximum of {TRACE_EVENT_NAME_MAX} characters
(which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). Successive calls to
this function with the same trace event type identifier and the same trace stream identifier shall
return the same event name.

The posix_trace_trid_eventid_open() function shall associate a user trace event name with a trace
event type identifier for a given trace stream. The trace stream is identified by the trid argument,
and it shall be an active trace stream. The trace event name is the string pointed to by the
argument event_name. It shall have a maximum of {TRACE_EVENT_NAME_MAX} characters
(which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). The number of user
trace event type identifiers that can be defined for any given process is limited by the maximum
value {TRACE_USER_EVENT_MAX}, which has the minimum value
{_POSIX_TRACE_USER_EVENT_MAX}.

If the Trace Inherit option is not supported, the posix_trace_trid_eventid_open() function shall
associate the user trace event name pointed to by the event_name argument with a trace event
type identifier that is unique for the process being traced in the trace stream identified by the trid
argument, and is returned in the variable pointed to by the event argument. If the user trace
event name has already been mapped for the traced process, then the previously assigned trace
event type identifier shall be returned. If the per-process user trace event name limit represented
by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined
POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 79)) user trace event shall be
returned.

If the Trace Inherit option is supported, the posix_trace_trid_eventid_open() function shall
associate the user trace event name pointed to by the event_name argument with a trace event
type identifier that is unique for all the processes being traced in the trace stream identified by
the trid argument, and is returned in the variable pointed to by the event argument. If the user trace
event name has already been mapped for the traced processes, then the previously assigned trace
event type identifier shall be returned. If the per-process user trace event name limit represented
by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined
**posix_trace_eventid_equal()**

POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 79)) user trace event shall be returned.

**RETURN VALUE**

Upon successful completion, the `posix_trace_eventid_get_name()` and `posix_trace_trid_eventid_open()` functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

The `posix_trace_eventid_equal()` function shall return a non-zero value if `event1` and `event2` are equal; otherwise, a value of zero shall be returned. No errors are defined. If either `event1` or `event2` are not valid trace event type identifiers for the trace stream specified by `trid` or if the `trid` is invalid, the behavior shall be unspecified.

The `posix_trace_eventid_get_name()` function stores the trace event name value in the object pointed to by `event_name`, if successful.

The `posix_trace_trid_eventid_open()` function stores the trace event type identifier value in the object pointed to by `event`, if successful.

**ERRORS**

The `posix_trace_eventid_get_name()` and `posix_trace_trid_eventid_open()` functions shall fail if:

- `[EINVAL]` The `trid` argument was not a valid trace stream identifier.
- `[EINVAL]` The trace event type identifier `event` was not associated with any name.
- `[ENAMETOOLONG]` The size of the name pointed to by the `event_name` argument was longer than the implementation-defined value `{TRACE_EVENT_NAME_MAX}`.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

Table 2-7 (on page 79), `posix_trace_event()`, `posix_trace_getnext_event()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<trace.h>`

**CHANGE HISTORY**


IEEE PASC Interpretations 1003.1 #123 and #129 are applied.
posix_trace_eventid_open()

NAME
posix_trace_eventid_open — trace functions for instrumenting application code (TRACING)

SYNOPSIS
```c
#include <sys/types.h>
#include <trace.h>

int posix_trace_eventid_open(const char *restrict event_name,
trace_event_id_t *restrict event_id);
```

DESCRIPTION
Refer to `posix_trace_event()`.
posix_trace_eventset_add( )

3017 SYNOPSIS
3018 #include <trace.h>
3019
3020 int posix_trace_eventset_add(trace_event_id_t event_id, trace_event_set_t *set);
3021 int posix_trace_eventset_del(trace_event_id_t event_id, trace_event_set_t *set);
3022 int posix_trace_eventset_empty(trace_event_set_t *set);
3023 int posix_trace_eventset_fill(trace_event_set_t *set, int what);
3024 int posix_trace_eventset_ismember(trace_event_id_t event_id, const trace_event_set_t *restrict set, int *restrict ismember);

DESCRIPTION
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.

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.

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.

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:

POSIX_TRACE_WOPID_EVENTS
All the process-independent implementation-defined system trace event types are included in the set.

POSIX_TRACE_SYSTEM_EVENTS
All the implementation-defined system trace event types are included in the set, as are those defined in IEEE Std 1003.1-2001.

POSIX_TRACE_ALL_EVENTS
All trace event types defined, both system and user, are included in the set.

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.

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.
RETURN VALUE
Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

ERRORS
These functions may fail if:

[EINVAL] The value of one of the arguments is invalid.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
posix_trace_set_filter(), posix_trace_trid_eventid_open(), the Base Definitions volume of IEEE Std 1003.1-2001, <trace.h>

CHANGE HISTORY
NAME
posix_trace_eventtypelist_getnext_id, posix_trace_eventtypelist_rewind — iterate over a
mapping of trace event types (TRACING)

SYNOPSIS
#include <trace.h>

int posix_trace_eventtypelist_getnext_id(trace_id_t trid,
  trace_event_id_t *restrict event, int *restrict unavailable);

int posix_trace_eventtypelist_rewind(trace_id_t trid);

DESCRIPTION
The first time posix_trace_eventtypelist_getnext_id() is called, the function shall return in the
variable pointed to by event the first trace event type identifier of the list of trace events of the
trace stream identified by the trid argument. Successive calls to
posix_trace_eventtypelist_getnext_id() return in the variable pointed to by event the next trace
event type identifier in that same list. Each time a trace event type identifier is successfully
written into the variable pointed to by the event argument, the variable pointed to by the
unavailable argument shall be set to zero. When no more trace event type identifiers are
available, and so none is returned, the variable pointed to by the unavailable argument shall be
set to a value different from zero.

The posix_trace_eventtypelist_rewind() function shall reset the next trace event type identifier to
be read to the first trace event type identifier from the list of trace events used in the trace stream
identified by trid.

RETURN VALUE
Upon successful completion, these functions shall return a value of zero. Otherwise, they shall
return the corresponding error number.

The posix_trace_eventtypelist_getnext_id() function stores the trace event type identifier value in
the object pointed to by event, if successful.

ERRORS
These functions shall fail if:

[EINVALL] The trid argument was not a valid trace stream identifier.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
posix_trace_event(), posix_trace_getnext_event(), posix_trace_trid_eventid_open(), the Base
Definitions volume of IEEE Std 1003.1-2001, <trace.h>

CHANGE HISTORY
IEEE PASC Interpretations 1003.1 #123 and #129 are applied.
posix_trace_flush()

NAME
posix_trace_flush — trace stream flush from a process (TRACING)

SYNOPSIS

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_flush(trace_id_t trid);
```

DESCRIPTION

Refer to posix_trace_create().
__NAME__
posix_trace_get_attr, posix_trace_get_status — retrieve the trace attributes or trace status (TRACING)

__SYNOPSIS__
```c
#include <trace.h>

int posix_trace_get_attr(trace_id_t trid, trace_attr_t *attr);
int posix_trace_get_status(trace_id_t trid, struct posix_trace_status_info *statusinfo);
```

__DESCRIPTION__
The `posix_trace_get_attr()` function shall copy the attributes of the active trace stream identified by `trid` into the object pointed to by the `attr` argument. If the Trace Log option is supported, `trid` may represent a pre-recorded trace log.

The `posix_trace_get_status()` function shall return, in the structure pointed to by the `statusinfo` argument, the current trace status for the trace stream identified by the `trid` argument. These status values returned in the structure pointed to by `statusinfo` shall have been appropriately read to ensure that the returned values are consistent. If the Trace Log option is supported and the `trid` argument refers to a pre-recorded trace stream, the status shall be the status of the completed trace stream.

Each time the `posix_trace_get_status()` function is used, the overrun status of the trace stream shall be reset to POSIX_TRACE_NO_OVERRUN immediately after the call completes. If the Trace Log option is supported, the `posix_trace_get_status()` function shall behave the same as 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()` function is used, the log overrun status of the trace stream shall be reset to POSIX_TRACE_NO_OVERRUN and the `flush_error` status shall be reset to zero immediately after the call completes.

- If the `trid` argument refers to a pre-recorded trace stream, the status returned shall be the status of the completed trace stream and the status values of the trace stream shall not be reset.

__RETURN VALUE__
Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

The `posix_trace_get_attr()` function stores the trace attributes in the object pointed to by `attr`, if successful.

The `posix_trace_get_status()` function stores the trace status in the object pointed to by `statusinfo`, if successful.

__ERRORS__
These functions shall fail if:

- [EINVAL] The trace stream argument `trid` does not correspond to a valid active trace stream or a valid trace log.
posix_trace_get_attr()

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
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>

CHANGE HISTORY
IEEE PASC Interpretation 1003.1 #123 is applied.
**NAME**

posix_trace_get_filter, posix_trace_set_filter — retrieve and set the filter of an initialized trace stream (TRACING)

**SYNOPSIS**

```c
#include <trace.h>

int posix_trace_get_filter(trace_id_t trid, trace_event_set_t *set);
int posix_trace_set_filter(trace_id_t trid, const trace_event_set_t *set, int how);
```

**DESCRIPTION**

The `posix_trace_get_filter()` function shall retrieve, into the argument pointed to by `set`, the actual trace event filter from the trace stream specified by `trid`.

The `posix_trace_set_filter()` function shall change the set of filtered trace event types after a trace stream identified by the `trid` argument is created. This function may be called prior to starting the trace stream, or while the trace stream is active. By default, if no call is made to `posix_trace_set_filter()`, all trace events shall be recorded (that is, none of the trace event types are filtered out).

If this function is called while the trace is in progress, a special system trace event, POSIX_TRACE_FILTER, shall be recorded in the trace indicating both the old and the new sets of filtered trace event types (see Table 2-4 (on page 78) and Table 2-6 (on page 79)).

If the `posix_trace_set_filter()` function is interrupted by a signal, an error shall be returned and the filter shall not be changed. In this case, the state of the trace stream shall not be changed.

The value of the argument `how` indicates the manner in which the set is to be changed and shall have one of the following values, as defined in the `<trace.h>` header:

- **POSIX_TRACE_SET_EVENTSET**
  - The resulting set of trace event types to be filtered shall be the trace event type set pointed to by the argument `set`.

- **POSIX_TRACE_ADD_EVENTSET**
  - The resulting set of trace event types to be filtered shall be the union of the current set and the trace event type set pointed to by the argument `set`.

- **POSIX_TRACE_SUB_EVENTSET**
  - The resulting set of trace event types to be filtered shall be all trace event types in the current set that are not in the set pointed to by the argument `set`; that is, remove each element of the specified set from the current filter.

**RETURN VALUE**

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

The `posix_trace_get_filter()` function stores the set of filtered trace event types in `set`, if successful.

**ERRORS**

These functions shall fail if:

- **[EINVAL]** The value of the `trid` argument does not correspond to an active trace stream or the value of the argument pointed to by `set` is invalid.

- **[EINTR]** The operation was interrupted by a signal.
posix_trace_get_filter()

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Table 2-4 (on page 78), Table 2-6 (on page 79), posix_trace_eventset_add(), the Base Definitions volume of IEEE Std 1003.1-2001, <trace.h>

CHANGE HISTORY

IEEE PASC Interpretation 1003.1 #123 is applied.
NAME
posix_trace_get_status — retrieve the trace status (TRACING)

SYNOPSIS
#include <trace.h>

int posix_trace_get_status(trace_id_t trid, 
struct posix_trace_status_info *statusinfo);

DESCRIPTION
Refer to posix_trace_get_attr().
NAME
posix_trace_getnext_event, posix_trace_timedgetnext_event, posix_trace_trygetnext_event —
retrieve a trace event (TRACING)

SYNOPSIS
#include <sys/types.h>
#include <trace.h>

int posix_trace_getnext_event(trace_id_t trid,
struct posix_trace_event_info *restrict event,
void *restrict data, size_t num_bytes,
size_t *restrict data_len, int *restrict unavailable);

int posix_trace_timedgetnext_event(trace_id_t trid,
struct posix_trace_event_info *restrict event,
void *restrict data, size_t num_bytes,
size_t *restrict data_len, int *restrict unavailable,
const struct timespec *restrict abs_timeout);

int posix_trace_trygetnext_event(trace_id_t trid,
struct posix_trace_event_info *restrict event,
void *restrict data, size_t num_bytes,
size_t *restrict data_len, int *restrict unavailable);

DESCRIPTION
The posix_trace_getnext_event() function shall report a recorded trace event either from an active
trace stream without log or a pre-recorded trace stream identified by the trid argument. The
posix_trace_trygetnext_event() function shall report a recorded trace event from an active trace
stream without log identified by the trid argument.

The trace event information associated with the recorded trace event shall be copied by the
function into the structure pointed to by the argument event and the data associated with the
trace event shall be copied into the buffer pointed to by the data argument.

The posix_trace_getnext_event() function shall block if the trid argument identifies an active trace
stream and there is currently no trace event ready to be retrieved. When returning, if a recorded
trace event was reported, the variable pointed to by the unavailable argument shall be set to zero.
Otherwise, the variable pointed to by the unavailable argument shall be set to a value different
from zero.

The posix_trace_timedgetnext_event() function shall attempt to get another trace event from an
active trace stream without log, as in the posix_trace_getnext_event() function. However, if no
trace event is available from the trace stream, the implied wait shall be terminated when the
timeout specified by the argument abs_timeout expires, and the function shall return the error
[ETIMEDOUT].

The timeout shall expire when the absolute time specified by abs_timeout passes, as measured by
the clock upon 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 passed at the time of the
call.

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 circumstance shall the function fail with a timeout if a trace event is immediately
available from the trace stream. The validity of the abs_timeout argument need not be checked if
posix_trace_getnext_event()

The behavior of this function for a pre-recorded trace stream is unspecified.

The posix_trace_trygetnext_event() function shall not block. This function shall return an error if the trid argument identifies a pre-recorded trace stream. If a recorded trace event was reported, the variable pointed to by the unavailable argument shall be set to zero. Otherwise, if no trace event was reported, the variable pointed to by the unavailable argument shall be set to a value different from zero.

The argument num_bytes shall be the size of the buffer pointed to by the data argument. The argument data_len reports to the application the length in bytes of the data record just transferred. If num_bytes is greater than or equal to the size of the data associated with the trace event pointed to by the event argument, all the recorded data shall be transferred. In this case, the truncation-status member of the trace event structure shall be either POSIX_TRACE_NOT_TRUNCATED, if the trace event data was recorded without truncation while tracing, or POSIX_TRACE_TRUNCATED_RECORD, if the trace event data was truncated when it was recorded. If the num_bytes argument is less than the length of recorded trace event data, the data transferred shall be truncated to a length of num_bytes, the value stored in the variable pointed to by data_len shall be equal to num_bytes, and the truncation-status member of the event structure argument shall be set to POSIX_TRACE_TRUNCATED_READ (see the posix_trace_event_info structure defined in <trace.h>).

The report of a trace event shall be sequential starting from the oldest recorded trace event. Trace events shall be reported in the order in which they were generated, up to an implementation-defined time resolution that causes the ordering of trace events occurring very close to each other to be unknown. Once reported, a trace event cannot be reported again from an active trace stream. Once a trace event is reported from an active trace stream without log, the trace stream shall make the resources associated with that trace event available to record future generated trace events.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

If successful, these functions store:

- The recorded trace event in the object pointed to by event
- The trace event information associated with the recorded trace event in the object pointed to by data
- The length of this trace event information in the object pointed to by data_len
- The value of zero in the object pointed to by unavailable

ERRORS

These functions shall fail if:

[EINVAL] The trace stream identifier argument trid is invalid.

The posix_trace_getnext_event() and posix_trace_timedgetnext_event() functions shall fail if:

[EINTR] The operation was interrupted by a signal, and so the call had no effect.

The posix_trace_trygetnext_event() function shall fail if:

[EINVAL] The trace stream identifier argument trid does not correspond to an active trace stream.
The `posix_trace_getnext_event()` function shall fail if:

- `[EINVAL]` There is no trace event immediately available from the trace stream, and the `timeout` argument is invalid.
- `[ETIMEDOUT]` No trace event was available from the trace stream before the specified `timeout` expired.

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`posix_trace_create()`, `posix_trace_open()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/types.h>`, `<trace.h>`

**CHANGE HISTORY**
IEEE PASC Interpretation 1003.1 #123 is applied.
NAME
posix_trace_open, posix_trace_rewind — trace log management (TRACING)

SYNOPSIS
```
#include <trace.h>

int posix_trace_open(int file_desc, trace_id_t *trid);
int posix_trace_rewind(trace_id_t trid);
```

DESCRIPTION
Refer to `posix_trace_close()`.
posix_trace_set_filter()

NAME
posix_trace_set_filter — set filter of an initialized trace stream (TRACING)

SYNOPSIS
#include <trace.h>

int posix_trace_set_filter(trace_id_t trid, const trace_event_set_t *set, int how);

DESCRIPTION
Refer to posix_trace_get_filter().
NAME

posix_trace_shutdown — trace stream shutdown from a process (TRACING)

SYNOPSIS

```c
#include <sys/types.h>
#include <trace.h>

int posix_trace_shutdown(trace_id_t trid);
```

DESCRIPTION

Refer to `posix_trace_create()`.
posix_trace_start()  

NAME
posix_trace_start, posix_trace_stop — trace start and stop (TRACING)

SYNOPSIS
#include <trace.h>

int posix_trace_start(trace_id_t trid);
int posix_trace_stop (trace_id_t trid);

DESCRIPTION
The posix_trace_start() and posix_trace_stop() functions, respectively, shall start and stop the trace stream identified by the argument trid.

The effect of calling the posix_trace_start() function shall be recorded in the trace stream as the POSIX_TRACE_START system trace event and the status of the trace stream shall become POSIX_TRACE_RUNNING. If the trace stream is in progress when this function is called, the POSIX_TRACE_START system trace event shall not be recorded and the trace stream shall continue to run. If the trace stream is full, the POSIX_TRACE_START system trace event shall not be recorded and the status of the trace stream shall not be changed.

The effect of calling the posix_trace_stop() function shall be recorded in the trace stream as the POSIX_TRACE_STOP system trace event and the status of the trace stream shall become POSIX_TRACE_SUSPENDED. If the trace stream is suspended when this function is called, the POSIX_TRACE_STOP system trace event shall not be recorded and the trace stream shall remain suspended. If the trace stream is full, the POSIX_TRACE_STOP system trace event shall not be recorded and the status of the trace stream shall not be changed.

RETURN VALUE
Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

ERRORS
These functions shall fail if:

[EINVAL] The value of the argument trid does not correspond to an active trace stream and thus no trace stream was started or stopped.

EINVAL

Examples
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
posix_trace_create(), the Base Definitions volume of IEEE Std 1003.1-2001, <trace.h>
System Interfaces

30531  **CHANGE HISTORY**
30533  IEEE PASC Interpretation 1003.1 #123 is applied.
NAME
posix_trace_timedgetnext_event, — retrieve a trace event (TRACING)

SYNOPSIS
#include <sys/types.h>
#include <trace.h>

int posix_trace_timedgetnext_event(trace_id_t trid,
struct posix_trace_event_info *restrict event,
void *restrict data, size_t num_bytes,
size_t *restrict data_len, int *restrict unavailable,
const struct timespec *restrict abs_timeout);

DESCRIPTION
Refer to posix_trace_getnext_event().
NAME

posix_trace_trid_eventid_open — open a trace event type identifier (TRACING)

SYNOPSIS

```c
#include <trace.h>

int posix_trace_trid_eventid_open(trace_id_t trid,
                                const char *restrict event_name,
                                trace_event_id_t *restrict event);
```

DESCRIPTION

Refer to `posix_trace_eventid_equal()`.
NAME
posix_trace_trygetnext_event — retrieve a trace event (TRACING)

SYNOPSIS
#include <sys/types.h>
#include <trace.h>

int posix_trace_trygetnext_event(trace_id_t trid,
struct posix_trace_event_info *restrict event,
void *restrict data, size_t num_bytes,
size_t *restrict data_len, int *restrict unavailable);

DESCRIPTION
Refer to posix_trace_getnext_event().
NAME
posix_typed_mem_get_info — query typed memory information (ADVANCED REALTIME)

SYNOPSIS
#include <sys/mman.h>

int posix_typed_mem_get_info(int fildes, structposix_typed_mem_info *info);

DESCRIPTION
The posix_typed_mem_get_info() function shall return, in the posix_tmi_length field of the
posix_typed_mem_info structure pointed to by info, the maximum length which may be
successfully allocated by the typed memory object designated by fildes. This maximum length
shall take into account the flag POSIX_TYPED_MEM_ALLOCATE or
POSIX_TYPED_MEM_ALLOCATE_CONTIG specified when the typed memory object
represented by fildes was opened. The maximum length is dynamic; therefore, the value returned
is valid only while the current mapping of the corresponding typed memory pool remains
unchanged.

If fildes represents a typed memory object opened with neither the
POSIX_TYPED_MEM_ALLOCATE flag nor the POSIX_TYPED_MEM_ALLOCATE_CONTIG
flag specified, the returned value of info->posix_tmi_length is unspecified.

The posix_typed_mem_get_info() function may return additional implementation-defined
information in other fields of the posix_typed_mem_info structure pointed to by info.

If the memory object specified by fildes is not a typed memory object, then the behavior of this
function is undefined.

RETURN VALUE
Upon successful completion, the posix_typed_mem_get_info() function shall return zero;
otherwise, the corresponding error status value shall be returned.

ERRORS
The posix_typed_mem_get_info() function shall fail if:

[EBADF] The fildes argument is not a valid open file descriptor.
[ENODEV] The fildes argument is not connected to a memory object supported by this
function.

This function shall not return an error code of [EINTR].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
An application that needs to allocate a block of typed memory with length dependent upon the
amount of memory currently available must either query the typed memory object to obtain the
amount available, or repeatedly invoke mmap() attempting to guess an appropriate length.
While the latter method is existing practice with malloc(), it is awkward and imprecise. The
posix_typed_mem_get_info() function allows an application to immediately determine available
memory. This is particularly important for typed memory objects that may in some cases be
scarce resources. Note that when a typed memory pool is a shared resource, some form of
mutual-exclusion or synchronization may be required while typed memory is being queried and
The existing *fstat(*) function is not suitable for this purpose. We realize that implementations may wish to provide other attributes of typed memory objects (for example, alignment requirements, page size, and so on). The *fstat(*) function returns a structure which is not extensible and, furthermore, contains substantial information that is inappropriate for typed memory objects.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

*fstat(*), *mmap(*), *posix_typed_mem_open(*), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/mman.h>

**CHANGE HISTORY**

NAME
posix_typed_mem_open — open a typed memory object (ADVANCED REALTIME)

SYNOPSIS
#include <sys/mman.h>

int posix_typed_mem_open(const char *name, int oflag, int tflag);

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:

POSIX_TYPED_MEM_ALLOCATE
Allocate on mmap().

POSIX_TYPED_MEM_ALLOCATE_CONTIG
Allocate contiguously on mmap().

POSIX_TYPED_MEM_MAP_ALLOCATABLE
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
posix_typed_mem_open() Returns a file descriptor for a typed memory object.

If successful, *posix_typed_mem_open()* shall return a file descriptor for the typed memory object that is the lowest numbered file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor shall not share it with any other processes. It is unspecified whether the file offset is set. The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared.

The behavior of *msync()* , *ftruncate()* , and all file operations other than *mmap()* , *posix_mem_offset()* , *posix_typed_mem_get_info()* , *fstat()* , *dup()* , *dup2()* , and *close()* , is unspecified when passed a file descriptor connected to a typed memory object by this function.

The file status flags of the open file description shall be set according to the value of *oflag*. Applications shall specify exactly one of the three access mode values described below and defined in the <fcntl.h> header, as the value of *oflag*.

O_RDONLY Open for read access only.
O_WRONLY Open for write access only.
O_RDWR Open for read or write access.

Upon successful completion, the *posix_typed_mem_open()* function shall return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it shall return −1 and set *errno* to indicate the error.

The *posix_typed_mem_open()* function shall fail if:

**[EACCES]** The typed memory object exists and the permissions specified by *oflag* are denied.

**[EINTR]** The *posix_typed_mem_open()* operation was interrupted by a signal.

**[EINVAL]** The flags specified in *tflag* are invalid (more than one of POSIX_TYPED_MEM_ALLOCATE, POSIX_TYPED_MEMALLOCATE_CONTIG, or POSIX_TYPED_MEM_MAP_ALLOCATABLE is specified).

**[EMFILE]** Too many file descriptors are currently in use by this process.

**[ENAMETOOLONG]** The length of the *name* argument exceeds [PATH_MAX] or a pathname component is longer than [NAME_MAX].

**[ENFILE]** Too many file descriptors are currently open in the system.

**[ENOENT]** The named typed memory object does not exist.

**[EPERM]** The caller lacks the appropriate privilege to specify the flag POSIX_TYPED_MEM_MAP_ALLOCATABLE in argument *tflag*. 


**posix_typed_mem_open()**

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
close(), dup(), exec, fcntl(), fstat(), ftruncate(), mmap(), msync(), posix_mem_offset(),
posix_typed_mem_get_info(), umask(), the Base Definitions volume of IEEE Std 1003.1-2001,
<fcntl.h>, <sys/mman.h>

**CHANGE HISTORY**
NAME
pow, powf, powl — power function

SYNOPSIS
#include <math.h>

double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);

DESCRIPTION
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.

These functions shall compute the value of \( x \) raised to the power \( y \), \( x^y \). If \( x \) is negative, the application shall ensure that \( y \) is an integer value.

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

RETURN VALUE
Upon successful completion, these functions shall return the value of \( x \) raised to the power \( y \).

- For finite values of \( x < 0 \), and finite non-integer values of \( y \), a domain error shall occur and either a NaN (if representable), or an implementation-defined value shall be returned.
- If the correct value would cause overflow, a range error shall occur and \text{pow}(), \text{powf}(), and \text{powl}() shall return \text{HUGE_VAL}, \text{HUGE_VALF}, and \text{HUGE_VALL}, respectively.
- 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.
- If \( x \) or \( y \) is a NaN, a NaN shall be returned (unless specified elsewhere in this description).
- For any value of \( y \) (including NaN), if \( x \) is +1, 1.0 shall be returned.
- For any value of \( x \) (including NaN), if \( y \) is ±0, 1.0 shall be returned.
- For any odd integer value of \( y > 0 \), if \( x \) is ±0, ±0 shall be returned.
- For \( y > 0 \) and not an odd integer, if \( x \) is ±0, +0 shall be returned.
- If \( x \) is −1, and \( y \) is ±Inf, 1.0 shall be returned.
- For \( |x| < 1 \), if \( y \) is −Inf, +Inf shall be returned.
- For \( |x| > 1 \), if \( y \) is −Inf, +0 shall be returned.
- For \( |x| < 1 \), if \( y \) is +Inf, +0 shall be returned.
- For \( |x| > 1 \), if \( y \) is +Inf, +Inf shall be returned.
- For \( y \) an odd integer < 0, if \( x \) is −Inf, −0 shall be returned.
- For \( y < 0 \) and not an odd integer, if \( x \) is −Inf, +0 shall be returned.
- For \( y \) an odd integer > 0, if \( x \) is −Inf, −Inf shall be returned.
- For \( y > 0 \) and not an odd integer, if \( x \) is −Inf, +Inf shall be returned.
For \( y < 0 \), if \( x \) is +Inf, +0 shall be returned.

For \( y > 0 \), if \( x \) is +Inf, +Inf shall be returned.

For \( y \) an odd integer < 0, 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.

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.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

**ERRORS**

These functions shall fail if:

**Domain Error** The value of \( x \) is negative and \( y \) is a finite non-integer.

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.

**Pole Error** The value of \( x \) is zero and \( y \) is negative.

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.

**Range Error** The result overflows.

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.

These functions may fail if:

**Range Error** The result underflows.

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.

**EXAMPLES**

None.

**APPLICATION USAGE**

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.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.
SEE ALSO

exp(), f 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>

CHANGE HISTORY

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

Issue 5

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

Issue 6

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The powf() and powl() functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.
NAME
pread — read from a file

SYNOPSIS
XSI
#include <unistd.h>
ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);

DESCRIPTION
Refer to read().

NAME
printf — print formatted output

SYNOPSIS
#include <stdio.h>
int printf(const char *restrict format, ...);

DESCRIPTION
Refer to fprintf().
NAME
pselect, select — synchronous I/O multiplexing

SYNOPSIS
#include <sys/select.h>

int pselect(int nfds, fd_set *restrict readfds,
    fd_set *restrict writefds, fd_set *restrict errorfds,
    const struct timespec *restrict timeout,
    const sigset_t *restrict sigmask);

int select(int nfds, fd_set *restrict readfds,
    fd_set *restrict writefds, fd_set *restrict errorfds,
    struct timeval *restrict timeout);

void FD_CLR(int fd, fd_set *fdset);

int FD_ISSET(int fd, fd_set *fdset);

void FD_SET(int fd, fd_set *fdset);

void FD_ZERO(fd_set *fdset);

DESCRIPTION

The pselect() function shall examine the file descriptor sets whose addresses are passed in the
readfds, writefds, and errorfds parameters to see whether some of their descriptors are ready for
reading, are ready for writing, or have an exceptional condition pending, respectively.

The select() function shall be equivalent to the pselect() function, except as follows:

- For the select() function, the timeout period is given in seconds and microseconds in an
  argument of type struct timeval, whereas for the pselect() function the timeout period is
  given in seconds and nanoseconds in an argument of type struct timespec.

- The select() function has no sigmask argument; it shall behave as pselect() does when sigmask
  is a null pointer.

- Upon successful completion, the select() function may modify the object pointed to by the
  timeout argument.

The pselect() and select() functions shall support regular files, terminal and pseudo-terminal
XSR devices, STREAMS-based files, FIFOs, pipes, and sockets. The behavior of pselect() and select()
on file descriptors that refer to other types of file is unspecified.

The nfds argument specifies the range of descriptors to be tested. The first nfds descriptors shall
be checked in each set; that is, the descriptors from zero through nfds–1 in the descriptor sets
shall be examined.

If the readfds argument is not a null pointer, it points to an object of type fd_set that on input
specifies the file descriptors to be checked for being ready to read, and on output indicates
which file descriptors are ready to read.

If the writefds argument is not a null pointer, it points to an object of type fd_set that on input
specifies the file descriptors to be checked for being ready to write, and on output indicates
which file descriptors are ready to write.

If the errorfds argument is not a null pointer, it points to an object of type fd_set that on input
specifies the file descriptors to be checked for error conditions pending, and on output indicates
which file descriptors have error conditions pending.

Upon successful completion, the pselect() or select() function shall modify the objects pointed to
by the readfds, writefds, and errorfds arguments to indicate which file descriptors are ready for
reading, ready for writing, or have an error condition pending, respectively, and shall return the
total number of ready descriptors in all the output sets. For each file descriptor less than nfds, the
corresponding bit shall be set on successful completion if it was set on input and the associated
condition is true for that file descriptor.

If none of the selected descriptors are ready for the requested operation, the `pselect()` or `select()`
function shall block until at least one of the requested operations becomes ready, until the
`timeout` occurs, or until interrupted by a signal. The `timeout` parameter controls how long the
`pselect()` or `select()` function shall take before timing out. If the `timeout` parameter is not a null
pointer, it specifies a maximum interval to wait for the selection to complete. If the specified
time interval expires without any requested operation becoming ready, the function shall return.

If the `timeout` parameter is a null pointer, then the call to `pselect()` or `select()` shall block
indefinitely until at least one descriptor meets the specified criteria. To effect a poll, the `timeout`
parameter should not be a null pointer, and should point to a zero-valued `timespec` structure.

The use of a timeout does not affect any pending timers set up by `alarm()`, `ualarm()`, or
`setitimer()`.

Implementations may place limitations on the maximum timeout interval supported. All
implementations shall support a maximum timeout interval of at least 31 days. If the `timeout`
argument specifies a timeout interval greater than the implementation-defined maximum value,
the maximum value shall be used as the actual timeout value. Implementations may also 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.

If `sigmask` is not a null pointer, then the `pselect()` function shall replace the signal mask of the
process by the set of signals pointed to by `sigmask` before examining the descriptors, and shall
restore the signal mask of the process before returning.

A descriptor shall be considered ready for reading when a call to an input function with
`O_NONBLOCK` clear would not block, whether or not the function would transfer data
successfully. (The function might return data, an end-of-file indication, or an error other than
one indicating that it is blocked, and in each of these cases the descriptor shall be considered
ready for reading.)

A descriptor shall be considered ready for writing when a call to an output function with
`O_NONBLOCK` clear would not block, whether or not the function would transfer data
successfully.

If a socket has a pending error, it shall be considered to have an exceptional condition pending.
Otherwise, what constitutes an exceptional condition is file type-specific. For a file descriptor for
use with a socket, it is protocol-specific except as noted below. For other file types it is
implementation-defined. If the operation is meaningless for a particular file type, `pselect()` or
`select()` shall indicate that the descriptor is ready for read or write operations, and shall indicate
that the descriptor has no exceptional condition pending.

If a descriptor refers to a socket, the implied input function is the `recvmsg()` function with
parameters requesting normal and ancillary data, such that the presence of either type shall
cause the socket to be marked as readable. The presence of out-of-band data shall be checked if
the socket option `SO_OOBINLINE` has been enabled, as out-of-band data is enqueued with
normal data. If the socket is currently listening, then it shall be marked as readable if an
incoming connection request has been received, and a call to the `accept()` function shall complete
without blocking.

If a descriptor refers to a socket, the implied output function is the `sendmsg()` function supplying
an amount of normal data equal to the current value of the `SO_SNDLOWAT` option for the
socket. If a non-blocking call to the `connect()` function has been made for a socket, and the
connection attempt has either succeeded or failed leaving a pending error, the socket shall be
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.

If the readfds, writefds, 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, writefds, 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.

File descriptors associated with regular files shall always select true for ready to read, ready to write, and error conditions.

On failure, the objects pointed to by the readfds, writefds, 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, writefds, and errorfds arguments shall have all bits set to 0.

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.

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.

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.

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.

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.

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.

RETURN VALUE

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.

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.

ERRORS

Under the following conditions, pselect() and select() shall fail and set errno to:

[EBADF] One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor.
pselect()

System Interfaces

30976  [EINTR] The function was interrupted before any of the selected events occurred and
30977 before the timeout interval expired.
30978 XSI If SA_RESTART has been set for the interrupting signal, it is implementation-
30979 defined whether the function restarts or returns with [EINTR].
30980 [EINVAL] An invalid timeout interval was specified.
30981 [EINVAL] The nfds argument is less than 0 or greater than FD_SETSIZE.
30982 XSR [EINVAL] One of the specified file descriptors refers to a STREAM or multiplexer that is
30983 linked (directly or indirectly) downstream from a multiplexer.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
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
as follows: the introduction of the pselect() function included the <sys/select.h> header and the
<sys/select.h> header defines all the related definitions for the pselect() and select() functions.
Backwards-compatibility to existing XSI implementations is handled by allowing <sys/time.h>
to include <sys/select.h>.

FUTURE DIRECTIONS
None.

SEE ALSO
accept(), alarm(), connect(), fcntl(), poll(), read(), recvmsg(), sendmsg(), setitimer(), ualarm(),
write(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/select.h>, <sys/time.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
In the ERRORS section, the text has been changed to indicate that [EINVAL] is returned when
nfds is less than 0 or greater than FD_SETSIZE. It previously stated less than 0, or greater than or
equal to FD_SETSIZE.
Text about timeout is moved from the APPLICATION USAGE section to the DESCRIPTION.

Issue 6
The Open Group Corrigendum U026/6 is applied, changing the occurrences of readfs and writefs
in the select() DESCRIPTION to be readfds and writefds.
Text referring to sockets is added to the DESCRIPTION.
The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are
marked as part of the XSI STREAMS Option Group.
The following new requirements on POSIX implementations derive from alignment with the
Single UNIX Specification:
• These functions are now mandatory.
The pselect() function is added for alignment with IEEE Std 1003.1g-2000 and additional detail
related to sockets semantics is added to the DESCRIPTION.
The `select()` function now requires inclusion of `<sys/select.h>`.

The `restrict` keyword is added to the `select()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME

pthread_atfork — register fork handlers

SYNOPSIS

```c
#include <pthread.h>

int pthread_atfork(void (*prepare)(void), void (*parent)(void),
                   void (*child)(void));
```

DESCRIPTION

The `pthread_atfork()` function shall declare fork handlers to be called before and after `fork()`, in the context of the thread that called `fork()`. The prepare fork handler shall be called before `fork()` processing commences. The parent fork handle shall be called after `fork()` processing completes in the parent process. The child fork handler shall be called after `fork()` processing completes in the child process. If no handling is desired at one or more of these three points, the corresponding fork handler address(es) may be set to NULL.

The order of calls to `pthread_atfork()` is significant. The parent and child fork handlers shall be called in the order in which they were established by calls to `pthread_atfork()`. The prepare fork handlers shall be called in the opposite order.

RETURN VALUE

Upon successful completion, `pthread_atfork()` shall return a value of zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The `pthread_atfork()` function shall fail if:

- [ENOMEM] Insufficient table space exists to record the fork handler addresses.

The `pthread_atfork()` function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

There are at least two serious problems with the semantics of `fork()` in a multi-threaded program. One problem has to do with state (for example, memory) covered by mutexes. Consider the case where one thread has a mutex locked and the state covered by that mutex is inconsistent while another thread calls `fork()`. In the child, the mutex is in the locked state (locked by a nonexistent thread and thus can never be unlocked). Having the child simply reinitialize the mutex is unsatisfactory since this approach does not resolve the question about how to correct or otherwise deal with the inconsistent state in the child.

It is suggested that programs that use `fork()` call an `exec` function very soon afterwards in the child process, thus resetting all states. In the meantime, only a short list of async-signal-safe library routines are promised to be available.

Unfortunately, this solution does not address the needs of multi-threaded libraries. Application programs may not be aware that a multi-threaded library is in use, and they feel free to call any number of library routines between the `fork()` and `exec` calls, just as they always have. Indeed, they may be extant single-threaded programs and cannot, therefore, be expected to obey new restrictions imposed by the threads library.
On the other hand, the multi-threaded library needs a way to protect its internal state during
fork() in case it is re-entered later in the child process. The problem arises especially in multi-
threaded I/O libraries, which are almost sure to be invoked between the fork() and exec calls to
effect I/O redirection. The solution may require locking mutex variables during fork(), or it may
entail simply resetting the state in the child after the fork() processing completes.

The pthread_atfork() function provides multi-threaded libraries with a means to protect
themselves from innocent application programs that call fork(), and it provides multi-threaded
application programs with a standard mechanism for protecting themselves from fork() calls in
a library routine or the application itself.

The expected usage is that the prepare handler acquires all mutex locks and the other two fork
handlers release them.

For example, an application can supply a prepare routine that acquires the necessary mutexes the
library maintains and supply child and parent routines that release those mutexes, thus ensuring
that the child gets a consistent snapshot of the state of the library (and that no mutexes are left
stranded). Alternatively, some libraries might be able to supply just a child routine that
reinitializes the mutexes in the library and all associated states to some known value (for
example, what it was when the image was originally executed).

When fork() is called, only the calling thread is duplicated in the child process. Synchronization
variables remain in the same state in the child as they were in the parent at the time fork() was
called. Thus, for example, mutex locks may be held by threads that no longer exist in the child
process, and any associated states may be inconsistent. The parent process may avoid this by
explicit code that acquires and releases locks critical to the child via pthread_atfork(). In addition,
any critical threads need to be recreated and reinitialized to the proper state in the child (also via
pthread_atfork()).

A higher-level package may acquire locks on its own data structures before invoking lower-level
packages. Under this scenario, the order specified for fork handler calls allows a simple rule of
initialization for avoiding package deadlock: a package initializes all packages on which it
depends before it calls the pthread_atfork() function for itself.

FUTURE DIRECTIONS
None.

SEE ALSO
atexit(), fork(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>

CHANGE HISTORY
First released in Issue 5. Derived from the POSIX Threads Extension.
IEEE PASC Interpretation 1003.1c #4 is applied.

Issue 6
The pthread_atfork() function is marked as part of the Threads option.
The <pthread.h> header is added to the SYNOPSIS.
**NAME**

pthread_attr_destroy, pthread_attr_init — destroy and initialize the thread attributes object

**SYNOPSIS**

```c
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_init(pthread_attr_t *attr);
```

**DESCRIPTION**

The `pthread_attr_destroy()` function shall destroy a thread attributes object. An implementation may cause `pthread_attr_destroy()` to set `attr` to an implementation-defined invalid value. A destroyed `attr` attributes object can be reinitialized using `pthread_attr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

The `pthread_attr_init()` function shall initialize a thread attributes object `attr` with the default value for all of the individual attributes used by a given implementation.

The resulting attributes object (possibly modified by setting individual attribute values) when used by `pthread_create()` defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to `pthread_create()`. Results are undefined if `pthread_attr_init()` is called specifying an already initialized `attr` attributes object.

**RETURN VALUE**

Upon successful completion, `pthread_attr_destroy()` and `pthread_attr_init()` shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

**ERRORS**

The `pthread_attr_init()` function shall fail if:

- `[ENOMEM]` Insufficient memory exists to initialize the thread attributes object.

These functions shall not return an error code of `[EINTR]`.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to support probable future standardization in these areas without requiring that the function itself be changed.

Attributes objects provide clean isolation of the configurable aspects of threads. For example, "stack size" is an important attribute of a thread, but it cannot be expressed portably. When porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects can help by allowing the changes to be isolated in a single place, rather than being spread across every instance of thread creation.

Attributes objects can be used to set up "classes’ of threads with similar attributes; for example, "threads with large stacks and high priority” or “threads with minimal stacks”. These classes can be defined in a single place and then referenced wherever threads need to be created. Changes to “class” decisions become straightforward, and detailed analysis of each `pthread_create()` call is not required.

The attributes objects are defined as opaque types as an aid to extensibility. If these objects had been specified as structures, adding new attributes would force recompilation of all multi-
threaded programs when the attributes objects are extended; this might not be possible if different program components were supplied by different vendors.

Additionally, opaque attributes objects present opportunities for improving performance. Argument validity can be checked once when attributes are set, rather than each time a thread is created. Implementations often need to cache kernel objects that are expensive to create. Opaque attributes objects provide an efficient mechanism to detect when cached objects become invalid due to attribute changes.

Since assignment is not necessarily defined on a given opaque type, implementation-defined default values cannot be defined in a portable way. The solution to this problem is to allow attributes objects to be initialized dynamically by attributes object initialization functions, so that default values can be supplied automatically by the implementation.

The following proposal was provided as a suggested alternative to the supplied attributes:

1. Maintain the style of passing a parameter formed by the bitwise-inclusive OR of flags to the initialization routines (pthread_create(), pthread_mutex_init(), pthread_cond_init()). The parameter containing the flags should be an opaque type for extensibility. If no flags are set in the parameter, then the objects are created with default characteristics. An implementation may specify implementation-defined flag values and associated behavior.

2. If further specialization of mutexes and condition variables is necessary, implementations may specify additional procedures that operate on the pthread_mutex_t and pthread_cond_t objects (instead of on attributes objects).

The difficulties with this solution are:

1. A bitmask is not opaque if bits have to be set into bitvector attributes objects using explicitly-coded bitwise-inclusive OR operations. If the set of options exceeds an int, application programmers need to know the location of each bit. If bits are set or read by encapsulation (that is, get and set functions), then the bitmask is merely an implementation of attributes objects as currently defined and should not be exposed to the programmer.

2. Many attributes are not Boolean or very small integral values. For example, scheduling policy may be placed in 3-bit or 4-bit, but priority requires 5-bit or more, thereby taking up at least 8 bits out of a possible 16 bits on machines with 16-bit integers. Because of this, the bitmask can only reasonably control whether particular attributes are set or not, and it cannot serve as the repository of the value itself. The value needs to be specified as a function parameter (which is non-extensible), or by setting a structure field (which is non-opaque), or by get and set functions (making the bitmask a redundant addition to the attributes objects).

Stack size is defined as an optional attribute because the very notion of a stack is inherently machine-dependent. Some implementations may not be able to change the size of the stack, for example, and others may not need to because stack pages may be discontiguous and can be allocated and released on demand.

The attribute mechanism has been designed in large measure for extensibility. Future extensions to the attribute mechanism or to any attributes object defined in this volume of IEEE Std 1003.1-2001 has to be done with care so as not to affect binary-compatibility.

Attributes objects, even if allocated by means of dynamic allocation functions such as malloc(), may have their size fixed at compile time. This means, for example, a pthread_create() in an implementation with extensions to pthread_attr_t cannot look beyond the area that the binary application assumes is valid. This suggests that implementations should maintain a size field in the attributes object, as well as possibly version information, if extensions in different directions
(possibly by different vendors) are to be accommodated.

FUTURE DIRECTIONS
None.

SEE ALSO
pthread_attr_getstackaddr(), pthread_attr_getstacksize(), pthread_attr_getdetachstate(),
pthread_create(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6
The pthread_attr_destroy() and pthread_attr_init() functions are marked as part of the Threads option.

IEEE PASC Interpretation 1003.1 #107 is applied, noting that the effect of initializing an already initialized thread attributes object is undefined.
NAME

pthread_attr_getdetachstate, pthread_attr_setdetachstate — get and set the detachstate attribute

SYNOPSIS

#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
int *detachstate);

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

DESCRIPTION

The detachstate attribute controls whether the thread is created in a detached state. If the thread
is created detached, then use of the ID of the newly created thread by the pthread_detach() or
pthread_join() function is an error.

The pthread_attr_getdetachstate() and pthread_attr_setdetachstate() functions, respectively, shall
get and set the detachstate attribute in the attr object.

For pthread_attr_getdetachstate(), detachstate shall be set to either
PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

For pthread_attr_setdetachstate(), the application shall set detachstate to either
PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

A value of PTHREAD_CREATE_DETACHED shall cause all threads created with attr to be in
the detached state, whereas using a value of PTHREAD_CREATE_JOINABLE shall cause all
threads created with attr to be in the joinable state. The default value of the detachstate attribute
shall be PTHREAD_CREATE_JOINABLE.

RETURN VALUE

Upon successful completion, pthread_attr_getdetachstate() and pthread_attr_setdetachstate() shall
return a value of 0; otherwise, an error number shall be returned to indicate the error.

The pthread_attr_getdetachstate() function stores the value of the detachstate attribute in detachstate
if successful.

ERRORS

The pthread_attr_setdetachstate() function shall fail if:

[EINVAL] The value of detachstate was not valid

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy(), pthread_attr_getstackaddr(), pthread_attr_getstacksize(), pthread_create(), the
Base Definitions volume of IEEE Std 1003.1-2001, < pthread.h >
**CHANGE HISTORY**

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

**Issue 6**

The `pthread_attr_setdetachstate()` and `pthread_attr_getdetachstate()` functions are marked as part of the Threads option.

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

pthread_attr_getguardsize, pthread_attr_setguardsize — get and set the thread guardsize attribute

SYNOPSIS

```c
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
    size_t *restrict guardsize);

int pthread_attr_setguardsize(pthread_attr_t *attr,
    size_t guardsize);
```

DESCRIPTION

The `pthread_attr_getguardsize()` function shall get the `guardsize` attribute in the `attr` object. This attribute shall be returned in the `guardsize` parameter.

The `pthread_attr_setguardsize()` function shall set the `guardsize` attribute in the `attr` object. The new value of this attribute shall be obtained from the `guardsize` parameter. If `guardsize` is zero, a guard area shall not be provided for threads created with `attr`. If `guardsize` is greater than zero, a guard area of at least size `guardsize` bytes shall be provided for each thread created with `attr`.

The `guardsize` attribute controls the size of the guard area for the created thread’s stack. The `guardsize` attribute provides protection against overflow of the stack pointer. If a thread’s stack is created with guard protection, the implementation allocates extra memory at the overflow end of the stack as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error shall result (possibly in a SIGSEGV signal being delivered to the thread).

A conforming implementation may round up the value contained in `guardsize` to a multiple of the configurable system variable `{PAGESIZE}` (see `<sys/mman.h>`). If an implementation rounds up the value of `guardsize` to a multiple of `{PAGESIZE}`, a call to `pthread_attr_getguardsize()` specifying `attr` shall store in the `guardsize` parameter the guard size specified by the previous `pthread_attr_setguardsize()` function call.

The default value of the `guardsize` attribute is `{PAGESIZE}` bytes. The actual value of `{PAGESIZE}` is implementation-defined.

If the `stackaddr` or `stack` attribute has been set (that is, the caller is allocating and managing its own thread stacks), the `guardsize` attribute shall be ignored and no protection shall be provided by the implementation. It is the responsibility of the application to manage stack overflow along with stack allocation and management in this case.

RETURN VALUE

If successful, the `pthread_attr_getguardsize()` and `pthread_attr_setguardsize()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The `pthread_attr_getguardsize()` and `pthread_attr_setguardsize()` functions shall fail if:

- [EINVAL] The attribute `attr` is invalid.
- [EINVAL] The parameter `guardsize` is invalid.

These functions shall not return an error code of [EINTR].
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
The `guardsize` attribute is provided to the application for two reasons:

1. Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and which knows its threads never overflow their stack, can save system resources by turning off guard areas.

2. When threads allocate large data structures on the stack, large guard areas may be needed to detect stack overflow.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`, `<sys/mman.h>`

CHANGE HISTORY
First released in Issue 5.

Issue 6
In the ERRORS section, a third [EINVAL] error condition is removed as it is covered by the second error condition.

The `restrict` keyword is added to the `pthread_attr_getguardsize()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME

pthread_attr_getinheritsched, pthread_attr_setinheritsched — get and set the inheritsched attribute (REALTIME THREADS)

SYNOPSIS

#include <pthread.h>

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

int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                int inheritsched);

DESCRIPTION

The pthread_attr_getinheritsched(), and pthread_attr_setinheritsched() functions, respectively, shall get and set the inheritsched attribute in the attr argument.

When the attributes objects are used by pthread_create(), the inheritsched attribute determines how the other scheduling attributes of the created thread shall be set.

PTHREAD_INHERIT_SCHED

Specifies that the thread scheduling attributes shall be inherited from the creating thread, 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.

The symbols PTHREAD_INHERIT_SCHED and PTHREAD_EXPLICIT_SCHED are defined in the <pthread.h> header.

The following thread scheduling attributes defined by IEEE Std 1003.1-2001 are affected by the inheritsched attribute: scheduling policy (schedpolicy), scheduling parameters (schedparam), and scheduling contention scope (contentionscope).

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.

The pthread_attr_setinheritsched() function may fail if:

- EINVAL: The value of inheritsched is not valid.
- ENOTSUP: An attempt was made to set the attribute to an unsupported value.

These functions shall not return an error code of [EINVAL].

EXAMPLES

None.

APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using pthread_create(). Using these routines does not affect the current running thread.

RATIONALE

None.
FUTURE DIRECTIONS
None.

SEE ALSO
pthread_attr_destroy(), pthread_attr_getscope(), pthread_attr_getschedpolicy(),
pthread_attr_getschedparam(), pthread_create(), the Base Definitions volume of
IEEE Std 1003.1-2001, <pthread.h>, <sched.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.
Marked as part of the Realtime Threads Feature Group.

Issue 6
The pthread_attr_getinheritsched() and pthread_attr_setinheritsched() functions are marked as part
of the Threads and Thread Execution Scheduling options.
The [ENOSYS] error condition has been removed as stubs need not be provided if an
implementation does not support the Thread Execution Scheduling option.
The restrict keyword is added to the pthread_attr_getinheritsched() prototype for alignment with
NAME

pthread_attr_getschedparam, pthread_attr_setschedparam — get and set the schedparam attribute

SYNOPSIS

THR

```c
#include <pthread.h>

int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
                     struct sched_param *restrict param);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
                     const struct sched_param *restrict param);
```

DESCRIPTION

The `pthread_attr_getschedparam()` and `pthread_attr_setschedparam()` functions, respectively, shall get and set the scheduling parameter attributes in the `attr` argument. The contents of the `param` structure are defined in the `<sched.h>` header. For the SCHED_FIFO and SCHED_RR policies, the only required member of `param` is `sched_priority`.

For the SCHED_SPORADIC policy, the required members of the `param` structure are `sched_priority`, `sched_ss_low_priority`, `sched_ss_repl_period`, `sched_ss_init_budget`, and `sched_ss_max_repl`. The specified `sched_ss_repl_period` must be greater than or equal to the specified `sched_ss_init_budget` for the function to succeed; if it is not, then the function shall fail.

The value of `sched_ss_max_repl` shall be within the inclusive range `[1,SS_REPL_MAX]` for the function to succeed; if not, the function shall fail.

RETURN VALUE

If successful, the `pthread_attr_getschedparam()` and `pthread_attr_setschedparam()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The `pthread_attr_setschedparam()` function may fail if:

- [EINVAL] The value of `param` is not valid.
- [ENOTSUP] An attempt was made to set the attribute to an unsupported value.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`pthread_attr_destroy()`, `pthread_attr_getscope()`, `pthread_attr_getinheritsched()`, `pthread_attr_getschedpolicy()`, ` pthread_create()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`, `<sched.h>`
THREAD/ATTR_GETSCHEDPARAM

CHANGE HISTORY

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

Issue 6

The pthread_attr_getschedparam() and pthread_attr_setschedparam() functions are marked as part of the Threads option.

The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

The restrict keyword is added to the pthread_attr_getschedparam() and pthread_attr_setschedparam() prototypes for alignment with the ISO/IEC 9899:1999 standard.
NAME
pthread_attr_getschedpolicy, pthread_attr_setschedpolicy — get and set the schedpolicy attribute (REALTIME THREADS)

SYNOPSIS
#include <pthread.h>

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
int *restrict policy);

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

DESCRIPTION
The pthread_attr_getschedpolicy() and pthread_attr_setschedpolicy() functions, respectively, shall get and set the schedpolicy attribute in the attr argument.

The supported values of policy shall include SCHED_FIFO, SCHED_RR, and SCHED_OTHER, which are defined in the <sched.h> header. When threads executing with the scheduling policy SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC are waiting on a mutex, they shall acquire the mutex in priority order when the mutex is unlocked.

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.

ERRORS
The pthread_attr_setschedpolicy() function may fail if:

[EINVAL] The value of policy is not valid.

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

These functions shall not return an error code of [EINTR].

EXAMPLES
None.

APPLICATION USAGE
After these attributes have been set, a thread can be created with the specified attributes using pthread_create(). Using these routines does not affect the current running thread.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
pthread_attr_destroy(), pthread_attr_getscope(), pthread_attr_getinheritsched(),
 pthread_attr_getschedparam(), pthread_create(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>, <sched.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.
The `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()` functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

The `restrict` keyword is added to the `pthread_attr_getschedpolicy()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
pthread_attr_getscope, pthread_attr_setscope — get and set the contentionscope attribute
(REALTIME THREADS)

SYNOPSIS
#include <pthread.h>

int pthread_attr_getscope(const pthread_attr_t *restrict attr,
                           int *restrict contentionscope);
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);

DESCRIPTION
The pthread_attr_getscope() and pthread_attr_setscope() functions, respectively, shall get and set
the contentionscope attribute in the attr object.

The contentionscope attribute may have the values PTHREAD_SCOPE_SYSTEM, signifying
system scheduling contention scope, or PTHREAD_SCOPE_PROCESS, signifying process
scheduling contention scope. The symbols PTHREAD_SCOPE_SYSTEM and
PTHREAD_SCOPE_PROCESS are defined in the <pthread.h> header.

RETURN VALUE
If successful, the pthread_attr_getscope() and pthread_attr_setscope() functions shall return zero;
otherwise, an error number shall be returned to indicate the error.

ERRORS
The pthread_attr_setscope() function may fail if:

EINVAL   The value of contentionscope is not valid.
ENOTSUP   An attempt was made to set the attribute to an unsupported value.

These functions shall not return an error code of [EINVAL].

EXAMPLES
None.

APPLICATION USAGE
After these attributes have been set, a thread can be created with the specified attributes using
pthread_create(). Using these routines does not affect the current running thread.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
pthread_attr_destroy(), pthread_attr_getinheritsched(), pthread_attr_getschedpolicy(),
pthread_attr_getschedparam(), pthread_create(), the Base Definitions volume of
IEEE Std 1003.1-2001, <pthread.h>, <sched.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.
Marked as part of the Realtime Threads Feature Group.
The `pthread_attr_getscope()` and `pthread_attr_setscope()` functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The `restrict` keyword is added to the `pthread_attr_getscope()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME

pthread_attr_getstack, pthread_attr_setstack — get and set stack attributes

SYNOPSIS

```c
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                        void **restrict stackaddr, size_t *restrict stacksize);

int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
                        size_t stacksize);
```

DESCRIPTION

The pthread_attr_getstack() and pthread_attr_setstack() functions, respectively, shall get and set
the thread creation stack attributes stackaddr and stacksize in the attr object.

The stack attributes specify the area of storage to be used for the created thread’s stack. The base
(lowest addressable byte) of the storage shall be stackaddr, and the size of the storage shall be
stacksize bytes. The stacksize shall be at least {PTHREAD_STACK_MIN}. The stackaddr shall be
aligned appropriately to be used as a stack; for example, pthread_attr_setstack() may fail with
[EINVAL] if (stackaddr & 0x7) is not 0. All pages within the stack described by stackaddr and
stacksize shall be both readable and writable by the thread.

RETURN VALUE

Upon successful completion, these functions shall return a value of 0; otherwise, an error
number shall be returned to indicate the error.

The pthread_attr_getstack() function shall store the stack attribute values in stackaddr and stacksize
if successful.

ERRORS

The pthread_attr_setstack() function shall fail if:

[EINVAL] The value of stacksize is less than {PTHREAD_STACK_MIN} or exceeds an
implementation-defined limit.

The pthread_attr_setstack() function may fail if:

[EINVAL] The value of stackaddr does not have proper alignment to be used as a stack, or
if (stackaddr + stacksize) lacks proper alignment.

[EACCES] The stack page(s) described by stackaddr and stacksize are not both readable
and writable by the thread.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

These functions are appropriate for use by applications in an environment where the stack for a
thread must be placed in some particular region of memory.

While it might seem that an application could detect stack overflow by providing a protected
page outside the specified stack region, this cannot be done portably. Implementations are free
to place the thread’s initial stack pointer anywhere within the specified region to accommodate
the machine’s stack pointer behavior and allocation requirements. Furthermore, on some
architectures, such as the IA-64, “overflow” might mean that two separate stack pointers
allocated within the region will overlap somewhere in the middle of the region.
pthread_attr_getstack()

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
pthread_attr_init(), pthread_attr_setdetachstate(), pthread_attr_setstacksize(), pthread_create(), the Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <pthread.h>

CHANGE HISTORY
First released in Issue 6. Developed as an XSI extension and brought into the BASE by IEEE PASC Interpretation 1003.1 #101.
# pthread_attr_getstackaddr()

## NAME

pthread_attr_getstackaddr, pthread_attr_setstackaddr — get and set the stackaddr attribute

## SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr, void **restrict stackaddr);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

## DESCRIPTION

The `pthread_attr_getstackaddr()` and `pthread_attr_setstackaddr()` functions, respectively, shall get and set the thread creation `stackaddr` attribute in the `attr` object.

The `stackaddr` attribute specifies the location of storage to be used for the created thread’s stack.

The size of the storage shall be at least `{PTHREAD_STACK_MIN}`.

## RETURN VALUE

Upon successful completion, `pthread_attr_getstackaddr()` and `pthread_attr_setstackaddr()` shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The `pthread_attr_getstackaddr()` function stores the `stackaddr` attribute value in `stackaddr` if successful.

## ERRORS

No errors are defined.

These functions shall not return an error code of `[EINVAL]`.

## EXAMPLES

None.

## 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.

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

`pthread_attr_destroy()`, `pthread_attr_getdetachstate()`, `pthread_attr_getstack()`,
`pthread_attr_getstacksize()`, `pthread_attr_setstack()`, `pthread_create()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<limits.h>`, `<pthread.h>`
CHANGE HISTORY

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

Issue 6

The `pthread_attr_getstackaddr()` and `pthread_attr_setstackaddr()` functions are marked as part of the Threads and Thread Stack Address Attribute options.

The `restrict` keyword is added to the `pthread_attr_getstackaddr()` prototype for alignment with the ISO/IEC 9899:1999 standard.

These functions are marked obsolescent.
NAME

pthread_attr_getstacksize, pthread_attr_setstacksize — get and set the stacksize attribute

SYNOPSIS

#include <pthread.h>

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

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

DESCRIPTION

The pthread_attr_getstacksize() and pthread_attr_setstacksize() functions, respectively, shall get and set the thread creation stacksize attribute in the attr object.

The stacksize attribute shall define the minimum stack size (in bytes) allocated for the created threads stack.

RETURN VALUE

Upon successful completion, pthread_attr_getstacksize() and pthread_attr_setstacksize() shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The pthread_attr_getstacksize() function stores the stacksize attribute value in stacksize if successful.

ERRORS

The pthread_attr_setstacksize() function shall fail if:

[EINVAL] The value of stacksize is less than [PTHREAD_STACK_MIN] or exceeds a system-imposed limit.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy(), pthread_attr_getstackaddr(), pthread_attr_getdetachstate(), pthread_create(), the Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <pthread.h>

CHANGE HISTORY

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

Issue 6

The pthread_attr_getstacksize() and pthread_attr_setstacksize() functions are marked as part of the Threads and Thread Stack Address Attribute options.

The restrict keyword is added to the pthread_attr_getstacksize() prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME

pthread_attr_init — initialize the thread attributes object

SYNOPSIS

```c
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```

DESCRIPTION

Refer to pthread_attr_destroy().
NAME
pthread_attr_setdetachstate — set the detachstate attribute

SYNOPSIS
THR

```c
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

DESCRIPTION
Refer to `pthread_attr_getdetachstate()`.
Name

pthread_attr_setguardsize — set the thread guardsize attribute

Synopsis

```c
#include <pthread.h>

int pthread_attr_setguardsize(pthread_attr_t *attr,
               size_t guardsize);
```

Description

Refer to `pthread_attr_getguardsize()`.
NAME

pthread_attr_setinheritsched — set the inheritsched attribute (REALTIME THREADS)

SYNOPSIS

#include <pthread.h>

int pthread_attr_setinheritsched(pthread_attr_t *attr,
int inheritsched);

DESCRIPTION

Refer to pthread_attr_getinheritsched().
NAME
pthread_attr_setschedparam — set the schedparam attribute

SYNOPSIS
THR

#include <pthread.h>

int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
const struct sched_param *restrict param);

DESCRIPTION
Refer to pthread_attr_getschedparam().
NAME
pthread_attr_setschedpolicy — set the schedpolicy attribute (REALTIME THREADS)

SYNOPSIS
#include <pthread.h>

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

DESCRIPTION
Refer to pthread_attr_getschedpolicy().
NAME
pthread_attr_setscope — set the contentionscope attribute (REALTIME THREADS)

SYNOPSIS
#include <pthread.h>

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);

DESCRIPTION
Refer to pthread_attr_getscope().
NAME

pthread_attr_setstack — set the stack attribute

SYNOPSIS

XSI

```c
#include <pthread.h>

int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
                          size_t stacksize);
```

DESCRIPTION

Refer to `pthread_attr_getstack()`.
NAME
pthread_attr_setstackaddr — set the stackaddr attribute

SYNOPSIS
THR TSA
#include <pthread.h>

OB
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);

DESCRIPTION
Refer to pthread_attr_getstackaddr().
NAME
pthread_attr_setstacksize — set the stacksize attribute

SYNOPSIS
#include <pthread.h>

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

DESCRIPTION
Refer to pthread_attr_getstacksize().
**pthread_barrier_destroy()**

**NAME**

pthread_barrier_destroy, pthread_barrier_init — destroy and initialize a barrier object
(ADVANCED REALTIME THREADS)

**SYNOPSIS**

```c
#include <pthread.h>

int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
const pthread_barrierattr_t *restrict attr, unsigned count);
```

**DESCRIPTION**

The `pthread_barrier_destroy()` function shall destroy the barrier referenced by `barrier` and release any resources used by the barrier. The effect of subsequent use of the barrier is undefined until the barrier is reinitialized by another call to `pthread_barrier_init()`. An implementation may use this function to set `barrier` to an invalid value. The results are undefined if `pthread_barrier_destroy()` is called when any thread is blocked on the barrier, or if this function is called with an uninitialized barrier.

The `pthread_barrier_init()` function shall allocate any resources required to use the barrier referenced by `barrier` and shall initialize the barrier with attributes referenced by `attr`. If `attr` is NULL, the default barrier attributes shall be used; the effect is the same as passing the address of a default barrier attributes object. The results are undefined if `pthread_barrier_init()` is called when any thread is blocked on the barrier (that is, has not returned from the `pthread_barrier_wait()` call). The results are undefined if a barrier is used without first being initialized. The results are undefined if `pthread_barrier_init()` is called specifying an already initialized barrier.

The `count` argument specifies the number of threads that must call `pthread_barrier_wait()` before any of them successfully return from the call. The value specified by `count` must be greater than zero.

If the `pthread_barrier_init()` function fails, the barrier shall not be initialized and the contents of `barrier` are undefined.

Only the object referenced by `barrier` may be used for performing synchronization. The result of referring to copies of that object in calls to `pthread_barrier_destroy()` or `pthread_barrier_wait()` is undefined.

**RETURN VALUE**

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

**ERRORS**

The `pthread_barrier_destroy()` function may fail if:

- **[EBUSY]** The implementation has detected an attempt to destroy a barrier while it is in use (for example, while being used in a `pthread_barrier_wait()` call) by another thread.

- **[EINVAL]** The value specified by `barrier` is invalid.

The `pthread_barrier_init()` function shall fail if:

- **[EINVAL]** The system lacks the necessary resources to initialize another barrier.

- **[EINVAL]** The value specified by `count` is equal to zero.
The `pthread_barrier_init()` function may fail if:

- `[ENOMEM]` Insufficient memory exists to initialize the barrier.
- `[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.
- `[EINVAL]` The value specified by `attr` is invalid.

These functions shall not return an error code of `[EINTR]`.

**EXAMPLES**
None.

**APPLICATION USAGE**
The `pthread_barrier_destroy()` and `pthread_barrier_init()` functions are part of the Barriers option and need not be provided on all implementations.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
- `pthread_barrier_wait()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

**CHANGE HISTORY**
NAME
pthread_barrier_wait — synchronize at a barrier (ADVANCED REALTIME THREADS)

SYNOPSIS
#include <pthread.h>

int pthread_barrier_wait(pthread_barrier_t *barrier);

DESCRIPTION
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.

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.

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().

The results are undefined if this function is called with an uninitialized barrier.

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.

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.

RETURN VALUE
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.

ERRORS
The pthread_barrier_wait() function may fail if:

[EINVAL] The value specified by barrier does not refer to an initialized barrier object.

This function shall not return an error code of [EINTR].

APPLICATION USAGE
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.

The pthread_barrier_wait() function is part of the Barriers option and need not be provided on all implementations.
RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
`pthread_barrier_destroy()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

CHANGE HISTORY

In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.
NAME
pthread_barrierattr_destroy, pthread_barrierattr_init — destroy and initialize the barrier
attributes object (ADVANCED REALTIME THREADS)

SYNOPSIS
#include <pthread.h>

int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
int pthread_barrierattr_init(pthread_barrierattr_t *attr);

DESCRIPTION
The pthread_barrierattr_destroy() function shall destroy a barrier attributes object. A destroyed
attr attributes object can be reinitialized using pthread_barrierattr_init(); the results of otherwise
referencing the object after it has been destroyed are undefined. An implementation may cause
pthread_barrierattr_destroy() to set the object referenced by attr to an invalid value.

The pthread_barrierattr_init() function shall initialize a barrier attributes object attr with the
default value for all of the attributes defined by the implementation.

Results are undefined if pthread_barrierattr_init() is called specifying an already initialized attr
attributes object.

After a barrier attributes object has been used to initialize one or more barriers, any function
affecting the attributes object (including destruction) shall not affect any previously initialized
barrier.

RETURN VALUE
If successful, the pthread_barrierattr_destroy() and pthread_barrierattr_init() functions shall return
zero; otherwise, an error number shall be returned to indicate the error.

ERRORS
The pthread_barrierattr_destroy() function may fail if:
EINVAL The value specified by attr is invalid.

The pthread_barrierattr_init() function shall fail if:
ENOMEM Insufficient memory exists to initialize the barrier attributes object.

These functions shall not return an error code of [EINTR].

EXAMPLES
None.

APPLICATION USAGE
The pthread_barrierattr_destroy() and pthread_barrierattr_init() functions are part of the Barriers
option and need not be provided on all implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
pthread_barrierattr_getpshared(), pthread_barrierattr_setpshared(), the Base Definitions volume of
IEEE Std 1003.1-2001, <pthread.h>.

In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.
NAME

pthread_barrierattr_getpshared, pthread_barrierattr_setpshared — get and set the process-shared attribute of the barrier attributes object (ADVANCED REALTIME THREADS)

SYNOPSIS

 Thr

#include <pthread.h>

 BAR TSH

int pthread_barrierattr_getpshared(const pthread_barrierattr_t * restrict attr, int *restrict pshared);

int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);

DESCRIPTION

The pthread_barrierattr_getpshared() function shall obtain the value of the process-shared attribute from the attributes object referenced by attr. The pthread_barrierattr_setpshared() function shall 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>.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

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.

ERRORS

These functions may fail if:

[EINVAL] The value specified by attr is invalid.

The pthread_barrierattr_setpshared() function may fail if:

[EINVAL] The new value specified for the process-shared attribute is not one of the legal values PTHREAD_PROCESS_SHARED or PTHREAD_PROCESS_PRIVATE.

These functions shall not return an error code of [EINVAL].
EXAMPLES
None.

APPLICATION USAGE
The `pthread_barrierattr_getpshared()` and `pthread_barrierattr_setpshared()` functions are part of the Barriers option and need not be provided on all implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
`pthread_barrier_destroy()`, `pthread_barrierattr_destroy()`, `pthread_barrierattr_init()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

CHANGE HISTORY
First released in Issue 6. Derived from IEEE Std 1003.1j-2000
NAME
pthread_barrierattr_init — initialize the barrier attributes object (ADVANCED REALTIME THREADS)

SYNOPSIS
#include <pthread.h>

```c
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

DESCRIPTION
Refer to pthread_barrierattr_destroy().
NAME
pthread_barrierattr_setpshared — set the process-shared attribute of the barrier attributes object
(ADVANCED REALTIME THREADS)

SYNOPSIS
#include <pthread.h>

int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
int pshared);

DESCRIPTION
Refer to pthread_barrierattr_getpshared().
NAME
pthread_cancel — cancel execution of a thread

SYNOPSIS
#include <pthread.h>

int pthread_cancel(pthread_t thread);

DESCRIPTION
The pthread_cancel() function shall request that thread be canceled. The target thread’s
cancelability state and type determines when the cancelation takes effect. When the cancelation
is acted on, the cancelation cleanup handlers for thread shall be called. When the last cancelation
cleanup handler returns, the thread-specific data destructor functions shall be called for thread.
When the last destructor function returns, thread shall be terminated.
The cancelation processing in the target thread shall run asynchronously with respect to the
calling thread returning from pthread_cancel().

RETURN VALUE
If successful, the pthread_cancel() function shall return zero; otherwise, an error number shall be
returned to indicate the error.

ERRORS
The pthread_cancel() function may fail if:
[ESRCH] No thread could be found corresponding to that specified by the given thread
ID.
The pthread_cancel() function shall not return an error code of [EINTR].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
Two alternative functions were considered for sending the cancelation notification to a thread.
One would be to define a new SIGCANCEL signal that had the cancelation semantics when
delivered; the other was to define the new pthread_cancel() function, which would trigger the
cancelation semantics.
The advantage of a new signal was that so much of the delivery criteria were identical to that
used when trying to deliver a signal that making cancelation notification a signal was seen as
consistent. Indeed, many implementations implement cancelation using a special signal. On the
other hand, there would be no signal functions that could be used with this signal except
pthread_kill(), and the behavior of the delivered cancelation signal would be unlike any
previously existing defined signal.
The benefits of a special function include the recognition that this signal would be defined
because of the similar delivery criteria and that this is the only common behavior between a
cancelation request and a signal. In addition, the cancelation delivery mechanism does not have
to be implemented as a signal. There are also strong, if not stronger, parallels with language
exception mechanisms than with signals that are potentially obscured if the delivery mechanism
is visibly closer to signals.
In the end, it was considered that as there were so many exceptions to the use of the new signal
with existing signals functions it would be misleading. A special function has resolved this
problem. This function was carefully defined so that an implementation wishing to provide the
cancellation functions on top of signals could do so. The special function also means that
implementations are not obliged to implement cancellation with signals.

FUTURE DIRECTIONS
None.

SEE ALSO
pthread_exit(), pthread_cond_timedwait(), pthread_join(), pthread_setcancelstate(), the Base
Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6
The pthread_cancel() function is marked as part of the Threads option.
NAME

pthread_cleanup_pop, pthread_cleanup_push — establish cancelation handlers

SYNOPSIS

THR

```
#include <pthread.h>

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

DESCRIPTION

The `pthread_cleanup_pop()` function shall remove the routine at the top of the calling thread’s cancelation cleanup stack and optionally invoke it (if `execute` is non-zero).

The `pthread_cleanup_push()` function shall push the specified cancelation cleanup handler `routine` onto the calling thread’s cancelation cleanup stack. The cancelation cleanup handler shall be popped from the cancelation cleanup stack and invoked with the argument `arg` when:

- The thread exits (that is, calls `pthread_exit()`).
- The thread acts upon a cancelation request.
- The thread calls `pthread_cleanup_pop()` with a non-zero `execute` argument.

These functions may be implemented as macros. The application shall ensure that they appear as statements, and in pairs within the same lexical scope (that is, the `pthread_cleanup_push()` macro may be thought to expand to a token list whose first token is `{` with `pthread_cleanup_pop()` expanding to a token list whose last token is the corresponding `}`).

The effect of calling `longjmp()` or `siglongjmp()` is undefined if there have been any calls to `pthread_cleanup_push()` or `pthread_cleanup_pop()` made without the matching call since the jump buffer was filled. The effect of calling `longjmp()` or `siglongjmp()` from inside a cancelation cleanup handler is also undefined unless the jump buffer was also filled in the cancelation cleanup handler.

RETURN VALUE

The `pthread_cleanup_push()` and `pthread_cleanup_pop()` functions shall not return a value.

ERRORS

No errors are defined.

These functions shall not return an error code of [EINVAL].

EXAMPLES

The following is an example using thread primitives to implement a cancelable, writers-priority read-write lock:

```
typedef struct {
    pthread_mutex_t lock;
    pthread_cond_t rcond,
    wcond;
    int lock_count; /* < 0 .. Held by writer. */
    /* > 0 .. Held by lock_count readers. */
    /* = 0 .. Held by nobody. */
    int waiting_writers; /* Count of waiting writers. */
 } rwlock;

void
waiting_reader_cleanup(void *arg)
{
```
System Interfaces

pthread_cleanup_pop()

rwlock *l;
1 = (rwlock *) arg;
pthread_mutex_unlock(&l->lock);
}

void
lock_for_read(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    pthread_cleanup_push(waiting_reader_cleanup, l);
    while ((l->lock_count < 0) && (l->waiting_writers != 0))
        pthread_cond_wait(&l->rcond, &l->lock);
    l->lock_count++;
    /*
    * Note the pthread_cleanup_pop executes
    * waiting_reader_cleanup.
    */
    pthread_cleanup_pop(1);
}

void
release_read_lock(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    if (--l->lock_count == 0)
        pthread_cond_signal(&l->wcond);
    pthread_mutex_unlock(&l->lock);
}

void
waiting_writer_cleanup(void *arg)
{
    rwlock *l;
    l = (rwlock *) arg;
    if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
        /*
        * This only happens if we have been canceled.
        */
        pthread_cond_broadcast(&l->wcond);
    }
    pthread_mutex_unlock(&l->lock);
}

void
lock_for_write(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    l->waiting_writers++;
    pthread_cleanup_push(waiting_writer_cleanup, l);
    while (l->lock_count != 0)
        pthread_cond_wait(&l->wcond, &l->lock);
    l->lock_count = -1;
    /*
* Note the pthread_cleanup_pop executes waiting_writer_cleanup.
*
pthread_cleanup_pop(1);
}

void
release_write_lock(rwlock *l)
{
pthread_mutex_lock(&l->lock);
  l->lock_count = 0;
  if (l->waiting_writers == 0)
    pthread_cond_broadcast(&l->rcond)
  else
    pthread_cond_signal(&l->wcond);
  pthread_mutex_unlock(&l->lock);
}

/*
* This function is called to initialize the read/write lock.
*/
void
initialize_rwlock(rwlock *l)
{
pthread_mutex_init(&l->lock, pthread_mutexattr_default);
  pthread_cond_init(&l->wcond, pthread_condattr_default);
  pthread_cond_init(&l->rcond, pthread_condattr_default);
  l->lock_count = 0;
  l->waiting_writers = 0;
}

reader_thread()
{
  lock_for_read(&lock);
  pthread_cleanup_push(release_read_lock, &lock);
  /*
   * Thread has read lock.
   */
  pthread_cleanup_pop(1);
}

writer_thread()
{
  lock_for_write(&lock);
  pthread_cleanup_push(release_write_lock, &lock);
  /*
   * Thread has write lock.
   */
  pthread_cleanup_pop(1);
}
APPLICATION USAGE

The two routines that push and pop cancelation cleanup handlers, `pthread_cleanup_push()` and `pthread_cleanup_pop()`, can be thought of as left and right parentheses. They always need to be matched.

RATIONALE

The restriction that the two routines that push and pop cancelation cleanup handlers, `pthread_cleanup_push()` and `pthread_cleanup_pop()`, have to appear in the same lexical scope allows for efficient macro or compiler implementations and efficient storage management. A sample implementation of these routines as macros might look like this:

```c
#define pthread_cleanup_push(rtn, arg) { 
  struct _pthread_handler_rec __cleanup_handler, **__head; 
  __cleanup_handler.rtn = rtn; 
  __cleanup_handler.arg = arg; 
  (void) pthread_getspecific(_pthread_handler_key, &__head); 
  __cleanup_handler.next = *__head; 
  *__head = &__cleanup_handler; 
} 
#define pthread_cleanup_pop(ex) 
  *__head = __cleanup_handler.next; 
  if (ex) (*__cleanup_handler.rtn)(__cleanup_handler.arg); 
}
```

A more ambitious implementation of these routines might do even better by allowing the compiler to note that the cancelation cleanup handler is a constant and can be expanded inline.

This volume of IEEE Std 1003.1-2001 currently leaves unspecified the effect of calling `longjmp()` from a signal handler executing in a POSIX System Interfaces function. If an implementation wants to allow this and give the programmer reasonable behavior, the `longjmp()` function has to call all cancelation cleanup handlers that have been pushed but not popped since the time `setjmp()` was called.

Consider a multi-threaded function called by a thread that uses signals. If a signal were delivered to a signal handler during the operation of `qsort()` and that handler were to call `longjmp()` (which, in turn, did not call the cancelation cleanup handlers) the helper threads created by the `qsort()` function would not be canceled. Instead, they would continue to execute and write into the argument array even though the array might have been popped off the stack.

Note that the specified cleanup handling mechanism is especially tied to the C language and, while the requirement for a uniform mechanism for expressing cleanup is language-independent, the mechanism used in other languages may be quite different. In addition, this mechanism is really only necessary due to the lack of a real exception mechanism in the C language, which would be the ideal solution.

There is no notion of a cancelation cleanup-safe function. If an application has no cancelation points in its signal handlers, blocks any signal whose handler may have cancelation points while calling async-unsafe functions, or disables cancelation while calling async-unsafe functions, all functions may be safely called from cancelation cleanup routines.

FUTURE DIRECTIONS

None.

SEE ALSO

`pthread_cancel()`, `pthread_setcancelstate()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`
**CHANGE HISTORY**

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

**Issue 6**

The `pthread_cleanup_pop()` and `pthread_cleanup_push()` functions are marked as part of the Threads option.

The APPLICATION USAGE section is added.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME

pthread_cond_broadcast, pthread_cond_signal — broadcast or signal a condition

SYNOPSIS
THR

#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);

DESCRIPTION

These functions shall unblock threads blocked on a condition variable.

The *pthread_cond_broadcast()* function shall unblock all threads currently blocked on the
specified condition variable *cond*.

The *pthread_cond_signal()* function shall unblock at least one of the threads that are blocked on
the specified condition variable *cond* (if any threads are blocked on *cond*).

If more than one thread is blocked on a condition variable, the scheduling policy shall determine
the order in which threads are unblocked. When each thread unblocked as a result of a
 pthread_cond_broadcast() or pthread_cond_signal() returns from its call to pthread_cond_wait() or
pthread_cond_timedwait(), the thread shall own the mutex with which it called
 pthread_cond_wait() or pthread_cond_timedwait(). The thread(s) that are unblocked shall contend
for the mutex according to the scheduling policy (if applicable), and as if each had called
 pthread_mutex_lock().

The *pthread_cond_broadcast()* or *pthread_cond_signal()* functions may be called by a thread
whether or not it currently owns the mutex that threads calling pthread_cond_wait() or
pthread_cond_timedwait() have associated with the condition variable during their waits;
however, if predictable scheduling behavior is required, then that mutex shall be locked by the
thread calling *pthread_cond_broadcast()* or *pthread_cond_signal()*.

The *pthread_cond_broadcast()* and *pthread_cond_signal()* functions shall have no effect if there are
no threads currently blocked on *cond*.

RETURN VALUE

If successful, the *pthread_cond_broadcast()* and *pthread_cond_signal()* functions shall return zero;
otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_cond_broadcast()* and *pthread_cond_signal()* function may fail if:

[EINVAL] The value *cond* does not refer to an initialized condition variable.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_cond_broadcast()* function is used whenever the shared-variable state has been
changed in a way that more than one thread can proceed with its task. Consider a single
producer/multiple consumer problem, where the producer can insert multiple items on a list
that is accessed one item at a time by the consumers. By calling the *pthread_cond_broadcast()*
function, the producer would notify all consumers that might be waiting, and thereby the
application would receive more throughput on a multi-processor. In addition,
 pthread_cond_broadcast() makes it easier to implement a read-write lock. The
pthread_cond_broadcast() function is needed in order to wake up all waiting readers when a
writer releases its lock. Finally, the two-phase commit algorithm can use this broadcast function
to notify all clients of an impending transaction commit.

It is not safe to use the `pthread_cond_signal()` function in a signal handler that is invoked
asynchronously. Even if it were safe, there would still be a race between the test of the Boolean
`pthread_cond_wait()` that could not be efficiently eliminated.

Mutexes and condition variables are thus not suitable for releasing a waiting thread by signaling
from code running in a signal handler.

RATIONALE

Multiple Awakenings by Condition Signal

On a multi-processor, it may be impossible for an implementation of `pthread_cond_signal()` to
avoid the unblocking of more than one thread blocked on a condition variable. For example,
consider the following partial implementation of `pthread_cond_wait()` and `pthread_cond_signal()`,
executed by two threads in the order given. One thread is trying to wait on the condition
variable, another is concurrently executing `pthread_cond_signal()`, while a third thread is already
waiting.

```
pthread_cond_wait(mutex, cond):
    value = cond->value; /* 1 */
    pthread_mutex_unlock(mutex); /* 2 */
    pthread_mutex_lock(cond->mutex); /* 10 */
    if (value == cond->value) { /* 11 */
        me->next_cond = cond->waiter;
        cond->waiter = me;
        pthread_mutex_unlock(cond->mutex);
        unable_to_run(me);
    } else
    pthread_mutex_unlock(cond->mutex); /* 12 */
    pthread_mutex_lock(mutex); /* 13 */

pthread_cond_signal(cond):
    pthread_mutex_lock(cond->mutex); /* 3 */
    cond->value++; /* 4 */
    if (cond->waiter) { /* 5 */
        sleeper = cond->waiter; /* 6 */
        cond->waiter = sleeper->next_cond; /* 7 */
        able_to_run(sleeper); /* 8 */
    }
    pthread_mutex_unlock(cond->mutex); /* 9 */
```

The effect is that more than one thread can return from its call to `pthread_cond_wait()` or
`pthread_cond_timedwait()` as a result of one call to `pthread_cond_signal()`. This effect is called
“spurious wakeup”. Note that the situation is self-correcting in that the number of threads that
are so awakened is finite; for example, the next thread to call `pthread_cond_wait()` after the
sequence of events above blocks.

While this problem could be resolved, the loss of efficiency for a fringe condition that occurs
only rarely is unacceptable, especially given that one has to check the predicate associated with a
condition variable anyway. Correcting this problem would unnecessarily reduce the degree of
concurrency in this basic building block for all higher-level synchronization operations.

An added benefit of allowing spurious wakeups is that applications are forced to code a
predicate-testing-loop around the condition wait. This also makes the application tolerate
superfluous condition broadcasts or signals on the same condition variable that may be coded in
some other part of the application. The resulting applications are thus more robust. Therefore,
IEEE Std 1003.1-2001 explicitly documents that spurious wakeups may occur.

FUTURE DIRECTIONS
None.

SEE ALSO
pthread_cond_destroy(), pthread_cond_timedwait(), the Base Definitions volume of
IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6
The pthread_cond_broadcast() and pthread_cond_signal() functions are marked as part of the
Threads option.
The APPLICATION USAGE section is added.
NAME

pthread_cond_destroy, pthread_cond_init — destroy and initialize condition variables

SYNOPSIS

THR

```c
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
const pthread_condattr_t *restrict attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

DESCRIPTION

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.

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.

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.

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.

Attempting to initialize an already initialized condition variable results in undefined behavior.

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.

RETURN VALUE

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.

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`.

ERRORS

The `pthread_cond_destroy()` function may fail if:

[EBUSY] The implementation has detected an attempt to destroy the object referenced by `cond` while it is referenced (for example, while being used in a `pthread_cond_wait()` or `pthread_cond_timedwait()`) by another thread.

[EINVAL] The value specified by `cond` is invalid.

The `pthread_cond_init()` function shall fail if:

[EAGAIN] The system lacked the necessary resources (other than memory) to initialize another condition variable.
Insufficient memory exists to initialize the condition variable.

The `pthread_cond_init()` function may fail if:

- **[ENOMEM]** The implementation has detected an attempt to reinitialize the object referenced by `cond`, a previously initialized, but not yet destroyed, condition variable.
- **[EINVAL]** The value specified by `attr` is invalid.

These functions shall not return an error code of [EINVAL].

**EXAMPLES**

A condition variable can be destroyed immediately after all the threads that are blocked on it are awakened. For example, consider the following code:

```c
struct list {
    pthread_mutex_t lm;
    ...
}

struct elt {
    key k;
    int busy;
    pthread_cond_t notbusy;
    ...
}

/* Find a list element and reserve it. */
struct elt *
list_find(struct list *lp, key k)
{
    struct elt *ep;
    pthread_mutex_lock(&lp->lm);
    while ((ep = find_elt(l, k) != NULL) && ep->busy)
        pthread_cond_wait(&ep->notbusy, &lp->lm);
    if (ep != NULL)
        ep->busy = 1;
    pthread_mutex_unlock(&lp->lm);
    return(ep);
}

delete_elt(struct list *lp, struct elt *ep)
{
    pthread_mutex_lock(&lp->lm);
    assert(ep->busy);
    ... remove ep from list ...
    ep->busy = 0; /* Paranoid. */
    (A) pthread_cond_broadcast(&ep->notbusy);
    pthread_mutex_unlock(&lp->lm);
    (B) pthread_cond_destroy(&ep->notbusy);
    free(ep);
}
```

In this example, the condition variable and its list element may be freed (line B) immediately after all threads waiting for it are awakened (line A), since the mutex and the code ensure that no other thread can touch the element to be deleted.
APPLICATION USAGE

None.

RATIONALE

See pthread_mutex_init(); a similar rationale applies to condition variables.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_broadcast(), pthread_cond_signal(), pthread_cond_timedwait(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The pthread_cond_destroy() and pthread_cond_init() functions are marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

The restrict keyword is added to the pthread_cond_init() prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME

pthread_cond_signal — signal a condition

SYNOPSIS

THR

```c
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
```

DESCRIPTION

Refer to `pthread_cond_broadcast()`.
NAME
pthread_cond_timedwait, pthread_cond_wait — wait on a condition

SYNOPSIS
THR
#include <pthread.h>
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                          pthread_mutex_t *restrict mutex,
                          const struct timespec *restrict abstime);

int pthread_cond_wait(pthread_cond_t *restrict cond,
                       pthread_mutex_t *restrict mutex);

DESCRIPTION
The pthread_cond_timedwait() and pthread_cond_wait() functions shall block on a condition variable. They shall be called with mutex locked by the calling thread or undefined behavior results.

These functions atomically release mutex and cause the calling thread to block on the condition variable cond; atomically here means ‘atomically with respect to access by another thread to the mutex and then the condition variable’. That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to pthread_cond_broadcast() or pthread_cond_signal() in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the pthread_cond_timedwait() or pthread_cond_wait() functions may occur. Since the return from pthread_cond_timedwait() or pthread_cond_wait() does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return.

The effect of using more than one mutex for concurrent pthread_cond_timedwait() or pthread_cond_wait() operations on the same condition variable is undefined; that is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding shall end when the wait returns.

A condition wait (whether timed or not) is a cancelation point. When the cancelability enable state of a thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a cancelation request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first cancelation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to pthread_cond_timedwait() or pthread_cond_wait(), but at that point notices the cancelation request and instead of returning to the caller of pthread_cond_timedwait() or pthread_cond_wait(), starts the thread cancelation activities, which includes calling cancelation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to pthread_cond_timedwait() or pthread_cond_wait() shall not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The pthread_cond_timedwait() function shall be equivalent to pthread_cond_wait(), except that an error is returned if the absolute time specified by abstime passes (that is, system time equals or exceeds abstime) before the condition cond is signaled or broadcasted, or if the absolute time specified by abstime has already been passed at the time of the call.
If the Clock Selection option is supported, the condition variable shall have a clock attribute which specifies the clock that shall be used to measure the time specified by the *abstime* argument. When such timeouts occur, *pthread_cond_timedwait()* shall nonetheless release and re-acquire the mutex referenced by *mutex*. The *pthread_cond_timedwait()* function is also a cancelation point.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it shall return zero due to spurious wakeup.

**RETURN VALUE**

Except in the case of [ETIMEDOUT], all these error checks shall act as if they were performed immediately at the beginning of processing for the function and shall cause an error return, in effect, prior to modifying the state of the mutex specified by *mutex* or the condition variable specified by *cond*.

Upon successful completion, a value of zero shall be returned; otherwise, an error number shall be returned to indicate the error.

**ERRORS**

The *pthread_cond_timedwait()* function shall fail if:

- [ETIMEDOUT] The time specified by *abstime* to *pthread_cond_timedwait()* has passed.
- The *pthread_cond_timedwait()* and *pthread_cond_wait()* functions may fail if:
  - [EINVAL] The value specified by *cond*, *mutex*, or *abstime* is invalid.
  - [EINVAL] Different mutexes were supplied for concurrent *pthread_cond_timedwait()* or *pthread_cond_wait()* operations on the same condition variable.
  - [EPERM] The mutex was not owned by the current thread at the time of the call.

These functions shall not return an error code of [EINTR].

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

**Condition Wait Semantics**

It is important to note that when *pthread_cond_wait()* and *pthread_cond_timedwait()* return without error, the associated predicate may still be false. Similarly, when *pthread_cond_timedwait()* 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.

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.

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.

It is thus recommended that a condition wait be enclosed in the equivalent of a “while loop” that checks the predicate.
Timed Wait Semantics

An absolute time measure was chosen for specifying the timeout parameter for two reasons.
First, a relative time measure can be easily implemented on top of a function that specifies
absolute time, but there is a race condition associated with specifying an absolute timeout on top
of a function that specifies relative timeouts. For example, assume that `clock_gettime()` returns
the current time and `cond_relative_timed_wait()` uses relative timeouts:

```c
clock_gettime(CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time -now;
cond_relative_timed_wait(c, m, &reltime);
```

If the thread is preempted between the first statement and the last statement, the thread blocks
for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout
also need not be recomputed if it is used multiple times in a loop, such as that enclosing a
condition wait.

For cases when the system clock is advanced discontinuously by an operator, it is expected that
implementations process any timed wait expiring at an intervening time as if that time had
actually occurred.

Cancellation and Condition Wait

A condition wait, whether timed or not, is a cancelation point. That is, the functions
`pthread_cond_wait()` or `pthread_cond_timedwait()` are points where a pending (or concurrent)
cancelation request is noticed. The reason for this is that an indefinite wait is possible at these
points—whatever event is being waited for, even if the program is totally correct, might never
occur; for example, some input data being awaited might never be sent. By making condition
wait a cancelation point, the thread can be canceled and perform its cancelation cleanup handler
even though it may be stuck in some indefinite wait.

A side effect of acting on a cancelation request while a thread is blocked on a condition variable
is to re-acquire the mutex before calling any of the cancelation cleanup handlers. This is done in
order to ensure that the cancelation cleanup handler is executed in the same state as the critical
code that lies both before and after the call to the condition wait function. This rule is also
required when interfacing to POSIX threads from languages, such as Ada or C++, which may
choose to map cancelation onto a language exception; this rule ensures that each exception
handler guarding a critical section can always safely depend upon the fact that the associated
mutex has already been locked regardless of exactly where within the critical section the
exception was raised. Without this rule, there would not be a uniform rule that exception
handlers could follow regarding the lock, and so coding would become very cumbersome.

Therefore, since some statement has to be made regarding the state of the lock when a
cancelation is delivered during a wait, a definition has been chosen that makes application
coding most convenient and error free.

When acting on a cancelation request while a thread is blocked on a condition variable, the
implementation is required to ensure that the thread does not consume any condition signals
directed at that condition variable if there are any other threads waiting on that condition
variable. This rule is specified in order to avoid deadlock conditions that could occur if these two
independent requests (one acting on a thread and the other acting on the condition variable)
were not processed independently.
Performance of Mutexes and Condition Variables

Mutexes are expected to be locked only for a few instructions. This practice is almost automatically enforced by the desire of programmers to avoid long serial regions of execution (which would reduce total effective parallelism).

When using mutexes and condition variables, one tries to ensure that the usual case is to lock the mutex, access shared data, and unlock the mutex. Waiting on a condition variable should be a relatively rare situation. For example, when implementing a read-write lock, code that acquires a read-lock typically needs only to increment the count of readers (under mutual-exclusion) and return. The calling thread would actually wait on the condition variable only when there is already an active writer. So the efficiency of a synchronization operation is bounded by the cost of mutex lock/unlock and not by condition wait. Note that in the usual case there is no context switch.

This is not to say that the efficiency of condition waiting is unimportant. Since there needs to be at least one context switch per Ada rendezvous, the efficiency of waiting on a condition variable is important. The cost of waiting on a condition variable should be little more than the minimal cost for a context switch plus the time to unlock and lock the mutex.

Features of Mutexes and Condition Variables

It had been suggested that the mutex acquisition and release be decoupled from condition wait. This was rejected because it is the combined nature of the operation that, in fact, facilitates realtime implementations. Those implementations can atomically move a high-priority thread between the condition variable and the mutex in a manner that is transparent to the caller. This can prevent extra context switches and provide more deterministic acquisition of a mutex when the waiting thread is signaled. Thus, fairness and priority issues can be dealt with directly by the scheduling discipline. Furthermore, the current condition wait operation matches existing practice.

Scheduling Behavior of Mutexes and Condition Variables

Synchronization primitives that attempt to interfere with scheduling policy by specifying an ordering rule are considered undesirable. Threads waiting on mutexes and condition variables are selected to proceed in an order dependent upon the scheduling policy rather than in some fixed order (for example, FIFO or priority). Thus, the scheduling policy determines which thread(s) are awakened and allowed to proceed.

Timed Condition Wait

The `pthread_cond_timedwait()` function allows an application to give up waiting for a particular condition after a given amount of time. An example of its use follows:

```c
(void) pthread_mutex_lock(&t.mn);
    t.waiters++;
    clock_gettime(CLOCK_REALTIME, &ts);
    ts.tv_sec += 5;
    rc = 0;
    while (! mypredicate(&t) && rc == 0)
        rc = pthread_cond_timedwait(&t.cond, &t.mn, &ts);
    t.waiters--;
if (rc == 0) setmystate(&t);
(void) pthread_mutex_unlock(&t.mn);
```
By making the timeout parameter absolute, it does not need to be recomputed each time the program checks its blocking predicate. If the timeout was relative, it would have to be recomputed before each call. This would be especially difficult since such code would need to take into account the possibility of extra wakeups that result from extra broadcasts or signals on the condition variable that occur before either the predicate is true or the timeout is due.

FUTURE DIRECTIONS
None.

SEE ALSO
pthread_cond_signal(), pthread_cond_broadcast(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6
The pthread_cond_timedwait() and pthread_cond_wait() functions are marked as part of the Threads option.

The Open Group Corrigendum U021/9 is applied, correcting the prototype for the pthread_cond_wait() function.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for the Clock Selection option.

The ERRORS section has an additional case for [EPERM] in response to IEEE PASC Interpretation 1003.1c #28.

The restrict keyword is added to the pthread_cond_timedwait() and pthread_cond_wait() prototypes for alignment with the ISO/IEC 9899:1999 standard.
NAME

pthread_condattr_destroy, pthread_condattr_init — destroy and initialize the condition variable attributes object

SYNOPSIS

THR

```c
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_init(pthread_condattr_t *attr);
```

DESCRIPTION

The `pthread_condattr_destroy()` function shall destroy a condition variable attributes object; the object becomes, in effect, uninitialized. An implementation may cause `pthread_condattr_destroy()` to set the object referenced by `attr` to an invalid value. A destroyed `attr` attributes object can be reinitialized using `pthread_condattr_init()`, the results of otherwise referencing the object after it has been destroyed are undefined.

The `pthread_condattr_init()` function shall initialize a condition variable attributes object `attr` with the default value for all of the attributes defined by the implementation.

Results are undefined if `pthread_condattr_init()` is called specifying an already initialized `attr` attributes object.

After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) shall not affect any previously initialized condition variables.

This volume of IEEE Std 1003.1-2001 requires two attributes, the `clock` attribute and the `process-shared` attribute.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

RETURN VALUE

If successful, the `pthread_condattr_destroy()` and `pthread_condattr_init()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The `pthread_condattr_destroy()` function may fail if:

- `[EINVAL]` The value specified by `attr` is invalid.

The `pthread_condattr_init()` function shall fail if:

- `[ENOMEM]` Insufficient memory exists to initialize the condition variable attributes object.

These functions shall not return an error code of `[EINTR]`.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

See `pthread_attr_init()` and `pthread_mutex_init()`.

A `process-shared` attribute has been defined for condition variables for the same reason it has been defined for mutexes.
pthread_condattr_destroy()

FUTURE DIRECTIONS
None.

SEE ALSO
pthread_attr_destroy(), pthread_cond_destroy(), pthread_condattr_getpshared(), pthread_create(),
pthread_mutex_destroy(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6
The pthread_condattr_destroy() and pthread_condattr_init() functions are marked as part of the Threads option.
NAME

pthread_condattr_getclock, pthread_condattr_setclock — get and set the clock selection condition variable attribute (ADVANCED REALTIME)

SYNOPSIS

```c
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *restrict attr, 
clockid_t *restrict clock_id);
int pthread_condattr_setclock(pthread_condattr_t *attr, 
clockid_t clock_id);
```

DESCRIPTION

The `pthread_condattr_getclock()` function shall obtain the value of the clock attribute from the attributes object referenced by attr. The `pthread_condattr_setclock()` function shall set the clock attribute in an initialized attributes object referenced by attr. If `pthread Condattr_setclock()` is called with a `clock_id` argument that refers to a CPU-time clock, the call shall fail.

The `clock` attribute is the clock ID of the clock that shall be used to measure the timeout service of `pthread_cond_timedwait()`. The default value of the `clock` attribute shall refer to the system clock.

RETURN VALUE

If successful, the `pthread_condattr_getclock()` function shall return zero and store the value of the clock attribute of `attr` into the object referenced by the `clock_id` argument. Otherwise, an error number shall be returned to indicate the error.

If successful, the `pthread_condattr_setclock()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

- `EINVAL` The value specified by `attr` is invalid.

The `pthread_condattr_setclock()` function may fail if:

- `EINVAL` The value specified by `clock_id` does not refer to a known clock, or is a CPU-time clock.

These functions shall not return an error code of `EINTR`.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`pthread_cond_destroy()`, `pthread_cond_timedwait()`, `pthread_condattr_destroy()`, `pthread_condattr_getpshared()` (on page 1041),1 `pthread_condattr_init()`, `pthread_condattr_setpshared()` (on page 1045),1 `pthread_create()`, `pthread_mutex_init()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`
pthread_condattr_getclock()
NAME
pthread_condattr_getpshared, pthread_condattr_setpshared — get and set the process-shared condition variable attributes

SYNOPSIS
#include <pthread.h>

int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
   int *restrict pshared);
int pthread_condattr_setpshared(pthread_condattr_t *attr,
   int pshared);

DESCRIPTION
The pthread_condattr_getpshared() function shall obtain the value of the process-shared attribute from the attributes object referenced by attr. The pthread_condattr_setpshared() function shall 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 condition variable to be operated upon by any thread that has access to the memory where the condition variable is allocated, even if the condition variable is allocated in memory that is shared by multiple processes. If the process-shared attribute is PTHREAD_PROCESS_PRIVATE, the condition variable shall only be operated upon by threads created within the same process as the thread that initialized the condition variable; if threads of differing processes attempt to operate on such a condition variable, the behavior is undefined. The default value of the attribute is PTHREAD_PROCESS_PRIVATE.

RETURN VALUE
If successful, the pthread_condattr_setpshared() function shall return zero; otherwise, an error number shall be returned to indicate the error.

If successful, the pthread_condattr_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.

ERRORS
The pthread_condattr_getpshared() and pthread_condattr_setpshared() functions may fail if:

[EINVAL] The value specified by attr is invalid.

The pthread_condattr_setpshared() function may fail if:

[EINVAL] The new value specified for the attribute is outside the range of legal values for that attribute.

These functions shall not return an error code of [EINTR].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.
**FUTURE DIRECTIONS**
None.

**SEE ALSO**
- `pthread_create()`, `pthread_cond_destroy()`, `pthread_condattr_destroy()`, `pthread_mutex_destroy()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

**CHANGE HISTORY**
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

**Issue 6**
The `pthread_condattr_getpshared()` and `pthread_condattr_setpshared()` functions are marked as part of the Threads and Thread Process-Shared Synchronization options.
The `restrict` keyword is added to the `pthread_condattr_getpshared()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME

pthread_condattr_init — initialize the condition variable attributes object

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);
```

DESCRIPTION

Refer to pthread_condattr_destroy().
NAME

pthread_condattr_setclock — set the clock selection condition variable attribute

SYNOPSIS

THR CS

```c
#include <pthread.h>

int pthread_condattr_setclock(pthread_condattr_t *attr, clockid_t clock_id);
```

DESCRIPTION

Refer to `pthread_condattr_getclock()`.
NAME

pthread_condattr_setpshared — set the process-shared condition variable attribute

SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);
```

DESCRIPTION

Refer to `pthread_condattr_getpshared()`.
NAME

pthread_create — thread creation

SYNOPSIS

THR

```c
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
        const pthread_attr_t *restrict attr,
        void *(*start_routine)(void*), void *restrict arg);
```

DESCRIPTION

The `pthread_create()` function shall create a new thread, with attributes specified by `attr`, within a process. If `attr` is NULL, the default attributes shall be used. If the attributes specified by `attr` are modified later, the thread’s attributes shall not be affected. Upon successful completion, `pthread_create()` shall store the ID of the created thread in the location referenced by `thread`.

The thread is created executing `start_routine` with `arg` as its sole argument. If the `start_routine` returns, the effect shall be as if there was an implicit call to `pthread_exit()` using the return value of `start_routine` as the exit status. Note that the thread in which `main()` was originally invoked differs from this. When it returns from `main()`, the effect shall be as if there was an implicit call to `exit()` using the return value of `main()` as the exit status.

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.

The floating-point environment shall be inherited from the creating thread.

If `pthread_create()` fails, no new thread is created and the contents of the location referenced by `thread` are undefined.

TCT

If `_POSIX_THREAD_CPUTIME` is defined, the new thread shall have a CPU-time clock accessible, and the initial value of this clock shall be set to zero.

RETURN VALUE

If successful, the `pthread_create()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The `pthread_create()` function shall fail if:

- [EAGAIN] The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process [PTHREAD_THREADS_MAX] would be exceeded.
- [EINVAL] The value specified by `attr` is invalid.
- [EPERM] The caller does not have appropriate permission to set the required scheduling parameters or scheduling policy.

The `pthread_create()` function shall not return an error code of [EINVAL].
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
A suggested alternative to `pthread_create()` would be to define two separate operations: create and start. Some applications would find such behavior more natural. Ada, in particular, separates the “creation” of a task from its “activation”.

Splitting the operation was rejected by the standard developers for many reasons:

- The number of calls required to start a thread would increase from one to two and thus place an additional burden on applications that do not require the additional synchronization. The second call, however, could be avoided by the additional complication of a start-up state attribute.

- An extra state would be introduced: “created but not started”. This would require the standard to specify the behavior of the thread operations when the target has not yet started executing.

- For those applications that require such behavior, it is possible to simulate the two separate steps with the facilities that are currently provided. The `start_routine()` can synchronize by waiting on a condition variable that is signaled by the start operation.

An Ada implementor can choose to create the thread at either of two points in the Ada program: when the task object is created, or when the task is activated (generally at a “begin”). If the first approach is adopted, the `start_routine()` needs to wait on a condition variable to receive the order to begin “activation”. The second approach requires no such condition variable or extra synchronization. In either approach, a separate Ada task control block would need to be created when the task object is created to hold rendezvous queues, and so on.

An extension of the preceding model would be to allow the state of the thread to be modified between the create and start. This would allow the thread attributes object to be eliminated. This has been rejected because:

- All state in the thread attributes object has to be able to be set for the thread. This would require the definition of functions to modify thread attributes. There would be no reduction in the number of function calls required to set up the thread. In fact, for an application that creates all threads using identical attributes, the number of function calls required to set up the threads would be dramatically increased. Use of a thread attributes object permits the application to make one set of attribute setting function calls. Otherwise, the set of attribute setting function calls needs to be made for each thread creation.

- Depending on the implementation architecture, functions to set thread state would require kernel calls, or for other implementation reasons would not be able to be implemented as macros, thereby increasing the cost of thread creation.

- The ability for applications to segregate threads by class would be lost.

Another suggested alternative uses a model similar to that for process creation, such as “thread fork”. The fork semantics would provide more flexibility and the “create” function can be implemented simply by doing a thread fork followed immediately by a call to the desired “start 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.

FUTURE DIRECTIONS
None.

SEE ALSO
fork(), pthread_exit(), pthread_join(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6
The pthread_create() function is marked as part of the Threads option.
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:
• The [EPERM] mandatory error condition is added.
The thread CPU-time clock semantics are added for alignment with IEEE Std 1003.1d-1999.
The restrict keyword is added to the pthread_create() prototype for alignment with the ISO/IEC 9899: 1999 standard.
The DESCRIPTION is updated to make it explicit that the floating-point environment is inherited from the creating thread.
NAME
pthread_detach — detach a thread

SYNOPSIS
THR #include <pthread.h>
int pthread_detach(pthread_t thread);

DESCRIPTION
The pthread_detach() function shall indicate to the implementation that storage for the thread
can be reclaimed when that thread terminates. If thread has not terminated,
pthread_detach() shall not cause it to terminate. The effect of multiple pthread_detach() calls on
the same target thread is unspecified.

RETURN VALUE
If the call succeeds, pthread_detach() shall return 0; otherwise, an error number shall be returned
to indicate the error.

ERRORS
The pthread_detach() function shall fail if:
[EINVAL] The implementation has detected that the value specified by thread does not
refer to a joinable thread.
[ESRCH] No thread could be found corresponding to that specified by the given thread
ID.

The pthread_detach() function shall not return an error code of [EINTR].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
The pthread_join() or pthread_detach() functions should eventually be called for every thread that
is created so that storage associated with the thread may be reclaimed.

It has been suggested that a “detach” function is not necessary; the detachstate thread creation
attribute is sufficient, since a thread need never be dynamically detached. However, need arises
in at least two cases:

1. In a cancelation handler for a pthread_join() it is nearly essential to have a pthread_detach()
function in order to detach the thread on which pthread_join() was waiting. Without it, it
would be necessary to have the handler do another pthread_join() to attempt to detach the
thread, which would both delay the cancelation processing for an unbounded period and
introduce a new call to pthread_join(), which might itself need a cancelation handler. A
dynamic detach is nearly essential in this case.

2. In order to detach the “initial thread” (as may be desirable in processes that set up server
threads).

FUTURE DIRECTIONS
None.
SEE ALSO

 pthread_join(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

The pthread_detach() function is marked as part of the Threads option.
NAME
pthread_equal — compare thread IDs

SYNOPSIS
THR
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);

DESCRIPTION
This function shall compare the thread IDs t1 and t2.

RETURN VALUE
The pthread_equal() function shall return a non-zero value if t1 and t2 are equal; otherwise, zero shall be returned.

If either t1 or t2 are not valid thread IDs, the behavior is undefined.

ERRORS
No errors are defined.

The pthread_equal() function shall not return an error code of [EINTR].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
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.

FUTURE DIRECTIONS
None.

SEE ALSO
pthread_create(), pthread_self(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6
The pthread_equal() function is marked as part of the Threads option.
NAME
pthread_exit — thread termination

SYNOPSIS
THR
#include <pthread.h>

void pthread_exit(void *value_ptr);

DESCRIPTION
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 cancelation 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 cancelation 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.

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.

The behavior of pthread_exit() is undefined if called from a cancelation cleanup handler or
destructor function that was invoked as a result of either an implicit or explicit call to
pthread_exit().

After a thread has terminated, the result of access to local (auto) variables of the thread is
undefined. Thus, references to local variables of the exiting thread should not be used for the
pthread_exit() value_ptr parameter value.

The process shall exit with an exit status of 0 after the last thread has been terminated. The
behavior shall be as if the implementation called exit() with a zero argument at thread
termination time.

RETURN VALUE
The pthread_exit() function cannot return to its caller.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
The normal mechanism by which a thread terminates is to return from the routine that was
specified in the pthread_create() call that started it. The pthread_exit() function provides the
capability for a thread to terminate without requiring a return from the start routine of that
thread, thereby providing a function analogous to exit().

Regardless of the method of thread termination, any cancelation cleanup handlers that have
been pushed and not yet popped are executed, and the destructors for any existing thread-
specific data are executed. This volume of IEEE Std 1003.1-2001 requires that cancelation
cleanup handlers be popped and called in order. After all cancelation cleanup handlers have
been executed, thread-specific data destructors are called, in an unspecified order, for each item
of thread-specific data that exists in the thread. This ordering is necessary because cancelation
cleanup handlers may rely on thread-specific data.

As the meaning of the status is determined by the application (except when the thread has been canceled, in which case it is PTHREAD_CANCELED), the implementation has no idea what an illegal status value is, which is why no address error checking is done.

FUTURE DIRECTIONS
None.

SEE ALSO
exit(), pthread_create(), pthread_join(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6
The pthread_exit() function is marked as part of the Threads option.
NAME

pthread_getconcurrency(), pthread_setconcurrency — get and set the level of concurrency

SYNOPSIS

XSI
#include <pthread.h>

int pthread_getconcurrency(void);
int pthread_setconcurrency(int new_level);

DESCRIPTION

Unbound threads in a process may or may not be required to be simultaneously active. By
default, the threads implementation ensures that a sufficient number of threads are active so that
the process can continue to make progress. While this conserves system resources, it may not
produce the most effective level of concurrency.

The pthread_setconcurrency() function allows an application to inform the threads
implementation of its desired concurrency level, new_level. The actual level of concurrency
provided by the implementation as a result of this function call is unspecified.

If new_level is zero, it causes the implementation to maintain the concurrency level at its
discretion as if pthread_setconcurrency() had never been called.

The pthread_getconcurrency() function shall return the value set by a previous call to the
pthread_setconcurrency() function. If the pthread_setconcurrency() function was not previously
called, this function shall return zero to indicate that the implementation is maintaining the
concurrency level.

A call to pthread_setconcurrency() shall inform the implementation of its desired concurrency
level. The implementation shall use this as a hint, not a requirement.

If an implementation does not support multiplexing of user threads on top of several kernel-
scheduled entities, the pthread_setconcurrency() and pthread_getconcurrency() functions are
provided for source code compatibility but they shall have no effect when called. To maintain
the function semantics, the new_level parameter is saved when pthread_setconcurrency() is called
so that a subsequent call to pthread_getconcurrency() shall return the same value.

RETURN VALUE

If successful, the pthread_setconcurrency() function shall return zero; otherwise, an error number
shall be returned to indicate the error.

The pthread_getconcurrency() function shall always return the concurrency level set by a previous
call to pthread_setconcurrency(). If the pthread_setconcurrency() function has never been called,
 pthread_getconcurrency() shall return zero.

ERRORS

The pthread_setconcurrency() function shall fail if:

[EINVAL] The value specified by new_level is negative.

[EAGAIN] The value specific by new_level would cause a system resource to be exceeded.

These functions shall not return an error code of [EINTR].
EXAMPLES
None.

APPLICATION USAGE
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 application's use of these
functions.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

CHANGE HISTORY
First released in Issue 5.
**NAME**

pthread_getcpuclockid — access a thread CPU-time clock (ADVANCED REALTIME THREADS)

**SYNOPSIS**

```
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);
```

**DESCRIPTION**

The `pthread_getcpuclockid()` function shall return in `clock_id` the clock ID of the CPU-time clock of the thread specified by `thread_id`, if the thread specified by `thread_id` exists.

**RETURN VALUE**

Upon successful completion, `pthread_getcpuclockid()` shall return zero; otherwise, an error number shall be returned to indicate the error.

**ERRORS**

The `pthread_getcpuclockid()` function may fail if:

- `[ESRCH]` The value specified by `thread_id` does not refer to an existing thread.

**EXAMPLES**

None.

**APPLICATION USAGE**

The `pthread_getcpuclockid()` function is part of the Thread CPU-Time Clocks option and need not be provided on all implementations.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`clock_getcpuclockid()`, `clock_getres()`, `timer_create()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`, `<time.h>`

**CHANGE HISTORY**


In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.
NAME

pthread_getschedparam, pthread_setschedparam — dynamic thread scheduling parameters

access (REALTIME THREADS)

SYNOPSIS

THR TPS

```c
#include <pthread.h>

int pthread_getschedparam(pthread_t thread, int *restrict policy,
                        struct sched_param *restrict param);

int pthread_setschedparam(pthread_t thread, int policy,
                        const struct sched_param *param);
```

DESCRIPTION

The `pthread_getschedparam()` and `pthread_setschedparam()` functions shall, respectively, get and set the scheduling policy and parameters of individual threads within a multi-threaded process to be retrieved and set. For SCHED_FIFO and SCHED_RR, the only required member of the `sched_param` structure is the priority `sched_priority`. For SCHED_OTHER, the affected scheduling parameters are implementation-defined.

The `pthread_getschedparam()` function shall retrieve the scheduling policy and scheduling parameters for the thread whose thread ID is given by `thread` and shall store those values in `policy` and `param`, respectively. The priority value returned from `pthread_getschedparam()` shall be the value specified by the most recent `pthread_setschedparam()`, `pthread_setschedprio()`, or `pthread_create()` call affecting the target thread. It shall not reflect any temporary adjustments to its priority as a result of any priority inheritance or ceiling functions. The `pthread_setschedparam()` function shall set the scheduling policy and associated scheduling parameters for the thread whose thread ID is given by `thread` to the policy and associated parameters provided in `policy` and `param`, respectively.

The `policy` parameter may have the value SCHED_OTHER, SCHED_FIFO, or SCHED_RR. The scheduling parameters for the SCHED_OTHER policy are implementation-defined. The SCHED_FIFO and SCHED_RR policies shall have a single scheduling parameter, `priority`.

If `_POSIX_THREAD_SPORADIC_SERVER` is defined, then the `policy` argument may have the value SCHED_SPORADIC, with the exception for the `pthread_setschedparam()` function that if the scheduling policy was not SCHED_SPORADIC at the time of the call, it is implementation-defined whether the function is supported; in other words, the implementation need not allow the application to dynamically change the scheduling policy to SCHED_SPORADIC. The sporadic server scheduling policy has the associated parameters `sched_ss_low_priority`, `sched_ss_repl_period`, `sched_ss_init_budget`, `sched_priority`, and `sched_ss_max_repl`. The specified `sched_ss_repl_period` shall be greater than or equal to the specified `sched_ss_init_budget` for the function to succeed; if it is not, then the function shall fail. The value of `sched_ss_max_repl` shall be within the inclusive range [1,[SS_REPL_MAX]] for the function to succeed; if not, the function shall fail.

If the `pthread_setschedparam()` function fails, the scheduling parameters shall not be changed for the target thread.

RETURN VALUE

If successful, the `pthread_getschedparam()` and `pthread_setschedparam()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.
The `pthread_getschedparam()` function may fail if:

- **ESRCH** The value specified by `thread` does not refer to an existing thread.

The `pthread_setschedparam()` function may fail if:

- **EINVAL** The value specified by `policy` or one of the scheduling parameters associated with the scheduling policy `policy` is invalid.
- **ENOTSUP** An attempt was made to set the policy or scheduling parameters to an unsupported value.
- **ENOTSUP** An attempt was made to dynamically change the scheduling policy to `SCHED_SPORADIC`, and the implementation does not support this change.
- **EPERM** The caller does not have the appropriate permission to set either the scheduling parameters or the scheduling policy of the specified thread.
- **EPERM** The implementation does not allow the application to modify one of the parameters to the value specified.
- **ESRCH** The value specified by `thread` does not refer to a existing thread.

These functions shall not return an error code of **EINTR**.

**EXCEPTIONS**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

- `pthread_setschedprio()`, `sched_getparam()`, `sched_getscheduler()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`, `<sched.h>`

**CHANGE HISTORY**

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

**Issue 6**

- The `pthread_getschedparam()` and `pthread_setschedparam()` functions are marked as part of the Threads and Thread Execution Scheduling options.
- The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.
- The Open Group Corrigendum U026/2 is applied, correcting the prototype for the `pthread_setschedparam()` function so that its second argument is of type `int`.
- The `SCHED_SPORADIC` scheduling policy is added for alignment with IEEE Std 1003.1d-1999.
- The `restrict` keyword is added to the `pthread_getschedparam()` prototype for alignment with the ISO/IEC 9899:1999 standard.
- The Open Group Corrigendum U047/1 is applied.
IEEE PASC Interpretation 1003.1 #96 is applied, noting that priority values can also be set by a call to the `pthread_setschedprio()` function.
NAME

pthread_getspecific, pthread_setspecific — thread-specific data management

SYNOPSIS

THR

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
```

DESCRIPTION

The `pthread_getspecific()` function shall return the value currently bound to the specified `key` on behalf of the calling thread.

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.

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.

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.

Both functions may be implemented as macros.

RETURN VALUE

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.

If successful, the `pthread_setspecific()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

No errors are returned from `pthread_getspecific()`.

The `pthread_setspecific()` function shall fail if:

[ENOMEM] Insufficient memory exists to associate the value with the key.

The `pthread_setspecific()` function may fail if:

[EINVAL] The key value is invalid.

These functions shall not return an error code of [EINVAL].
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
Performance and ease-of-use of `pthread_getspecific()` are critical for functions that rely on maintaining state in thread-specific data. Since no errors are required to be detected by it, and since the only error that could be detected is the use of an invalid key, the function to `pthread_getspecific()` has been designed to favor speed and simplicity over error reporting.

FUTURE DIRECTIONS
None.

SEE ALSO
`pthread_key_create()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

**Issue 6**
The `pthread_getspecific()` and `pthread_setspecific()` functions are marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #3 (Part 6) is applied, updating the DESCRIPTION.
NAME

pthread_join — wait for thread termination

SYNOPSIS

```c
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

DESCRIPTION

The `pthread_join()` function shall suspend execution of the calling thread until the target `thread` terminates, unless the target `thread` has already terminated. On return from a successful `pthread_join()` call with a non-NULL `value_ptr` argument, the value passed to `pthread_exit()` by the terminating thread shall be made available in the location referenced by `value_ptr`. When a `pthread_join()` returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to `pthread_join()` specifying the same target thread are undefined. If the thread calling `pthread_join()` is canceled, then the target thread shall not be detached.

It is unspecified whether a thread that has exited but remains unjoined counts against `{PTHREAD_THREADS_MAX}`.

RETURN VALUE

If successful, the `pthread_join()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The `pthread_join()` function shall fail if:

- `[EINVAL]` The implementation has detected that the value specified by `thread` does not refer to a joinable thread.
- `[ESRCH]` No thread could be found corresponding to that specified by the given thread ID.

The `pthread_join()` function may fail if:

- `[EDEADLK]` A deadlock was detected or the value of `thread` specifies the calling thread.

The `pthread_join()` function shall not return an error code of `[EINTR]`.

EXAMPLES

An example of thread creation and deletion follows:

```c
typedef struct {
    int *ar;
    long n;
} subarray;

void *
incer(void *arg)
{
    long i;
    for (i = 0; i < ((subarray *)arg)->n; i++)
        ((subarray *)arg)->ar[i]++;
}

int main(void)
{
    int ar[1000000];
```
The **pthread_join()** function is a convenience that has proven useful in multi-threaded applications. It is true that a programmer could simulate this function if it were not provided by passing extra state as part of the argument to the *start_routine*. The terminating thread would set a flag to indicate termination and broadcast a condition that is part of that state; a joining thread would wait on that condition variable. While such a technique would allow a thread to wait on more complex conditions (for example, waiting for multiple threads to terminate), waiting on individual thread termination is considered widely useful. Also, including the **pthread_join()** function in no way precludes a programmer from coding such complex waits. Thus, while not a primitive, including **pthread_join()** in this volume of IEEE Std 1003.1-2001 was considered valuable.

The **pthread_join()** function provides a simple mechanism allowing an application to wait for a thread to terminate. After the thread terminates, the application may then choose to clean up resources that were used by the thread. For instance, after **pthread_join()** returns, any application-provided stack storage could be reclaimed.

The **pthread_join()** or **pthread_detach()** function should eventually be called for every thread that is created with the *detachstate* attribute set to PTHREAD_CREATE_JOINABLE so that storage associated with the thread may be reclaimed.

The interaction between **pthread_join()** and cancelation is well-defined for the following reasons:

- The **pthread_join()** function, like all other non-async-cancel-safe functions, can only be called with deferred cancelability type.

- Cancelation cannot occur in the disabled cancelability state.

Thus, only the default cancelability state need be considered. As specified, either the **pthread_join()** call is canceled, or it succeeds, but not both. The difference is obvious to the application, since either a cancelation handler is run or **pthread_join()** returns. There are no race conditions since **pthread_join()** was called in the deferred cancelability state.

**FUTURE DIRECTIONS**

None.
SEE ALSO

pthread_create(), wait(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

The pthread_join() function is marked as part of the Threads option.
NAME

pthread_key_create — thread-specific data key creation

SYNOPSIS

THR

#include <pthread.h>

int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));

DESCRIPTION

The pthread_key_create() function shall create a thread-specific data key visible to all threads in
the process. Key values provided by pthread_key_create() are opaque objects used to locate
thread-specific data. Although the same key value may be used by different threads, the values
bound to the key by pthread_setspecific() are maintained on a per-thread basis and persist for the
life of the calling thread.

Upon key creation, the value NULL shall be associated with the new key in all active threads.
Upon thread creation, the value NULL shall be associated with all defined keys in the new
thread.

An optional destructor function may be associated with each key value. At thread exit, if a key
value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with
that key, the value of the key is set to NULL, and then the function pointed to is called with the
previously associated value as its sole argument. The order of destructor calls is unspecified if
more than one destructor exists for a thread when it exits.

If, after all the destructors have been called for all non-NULL values with associated destructors,
there are still some non-NULL values with associated destructors, then the process is repeated.
If, after at least [PTHREAD_DESTRUCTOR_ITERATIONS] iterations of destructor calls for
outstanding non-NULL values, there are still some non-NULL values with associated
destructors, implementations may stop calling destructors, or they may continue calling
destructors until no non-NULL values with associated destructors exist, even though this might
result in an infinite loop.

RETURN VALUE

If successful, the pthread_key_create() function shall store the newly created key value at *key and
shall return zero. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The pthread_key_create() function shall fail if:

[EAGAIN] The system lacked the necessary resources to create another thread-specific
data key, or the system-imposed limit on the total number of keys per process
[PTHREAD_KEYS_MAX] has been exceeded.

[ENOMEM] Insufficient memory exists to create the key.

The pthread_key_create() function shall not return an error code of [EINTR].
The following example demonstrates a function that initializes a thread-specific data key when it is first called, and associates a thread-specific object with each calling thread, initializing this object when necessary.

```c
static pthread_key_t key;
static pthread_once_t key_once = PTHREAD_ONCE_INIT;
static void
make_key()
{
    (void) pthread_key_create(&key, NULL);
}
func()
{
    void *ptr;
    (void) pthread_once(&key_once, make_key);
    if ((ptr = pthread_getspecific(key)) == NULL) {
        ptr = malloc(OBJECT_SIZE);
        ...
        (void) pthread_setspecific(key, ptr);
    }
    ...
}
```

Note that the key has to be initialized before `pthread_getspecific()` or `pthread_setspecific()` can be used. The `pthread_key_create()` call could either be explicitly made in a module initialization routine, or it can be done implicitly by the first call to a module as in this example. Any attempt to use the key before it is initialized is a programming error, making the code below incorrect.

```c
static pthread_key_t key;
func()
{
    void *ptr;
    /* KEY NOT INITIALIZED!!! THIS WON’T WORK!!! */
    if ((ptr = pthread_getspecific(key)) == NULL &&
        pthread_setspecific(key, NULL) != 0) {
        pthread_key_create(&key, NULL);
        ...
    }
}
```

**APPLICATION USAGE**

None.
RATIONALE

Destructor Functions

Normally, the value bound to a key on behalf of a particular thread is a pointer to storage allocated dynamically on behalf of the calling thread. The destructor functions specified with

pthread_key_create() are intended to be used to free this storage when the thread exits. Thread
cancellation cleanup handlers cannot be used for this purpose because thread-specific data may
persist outside the lexical scope in which the cancellation cleanup handlers operate.

If the value associated with a key needs to be updated during the lifetime of the thread, it may
be necessary to release the storage associated with the old value before the new value is bound.

Although the pthread_setspecific() function could do this automatically, this feature is not needed
often enough to justify the added complexity. Instead, the programmer is responsible for freeing
the stale storage:

```c
pthread_getspecific(key, &old);
new = allocate();
destructor(old);
pthread_setspecific(key, new);
```

Note: The above example could leak storage if run with asynchronous cancelation enabled. No such
problems occur in the default cancelation state if no cancelation points occur between the get
and set.

There is no notion of a destructor-safe function. If an application does not call pthread_exit()
from a signal handler, or if it blocks any signal whose handler may call pthread_exit() while
calling async-unsafe functions, all functions may be safely called from destructors.

Non-Idempotent Data Key Creation

There were requests to make pthread_key_create() idempotent with respect to a given key address
parameter. This would allow applications to call pthread_key_create() multiple times for a given
key address and be guaranteed that only one key would be created. Doing so would require the
key value to be previously initialized (possibly at compile time) to a known null value and
would require that implicit mutual-exclusion be performed based on the address and contents of
the key parameter in order to guarantee that exactly one key would be created.

Unfortunately, the implicit mutual-exclusion would not be limited to only pthread_key_create().
On many implementations, implicit mutual-exclusion would also have to be performed by
pthread_getspecific() and pthread_setspecific() in order to guard against using incompletely stored
or not-yet-visible key values. This could significantly increase the cost of important operations,
particularly pthread_getspecific().

Thus, this proposal was rejected. The pthread_key_create() function performs no implicit
synchronization. It is the responsibility of the programmer to ensure that it is called exactly once
per key before use of the key. Several straightforward mechanisms can already be used to
accomplish this, including calling explicit module initialization functions, using mutexes, and
using pthread_once(). This places no significant burden on the programmer, introduces no
possibly confusing ad hoc implicit synchronization mechanism, and potentially allows
commonly used thread-specific data operations to be more efficient.

FUTURE DIRECTIONS

None.
SEE ALSO

pthread_getspecific(), pthread_key_delete(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6
The pthread_key_create() function is marked as part of the Threads option.
IEEE PASC Interpretation 1003.1c #8 is applied, updating the DESCRIPTION.
NAME
pthread_key_delete — thread-specific data key deletion

SYNOPSIS
THR
#include <pthread.h>

int pthread_key_delete(pthread_key_t key);

DESCRIPTION
The pthread_key_delete() function shall delete a thread-specific data key previously returned by
pthread_key_create(). The thread-specific data values associated with key need not be NULL at
the time pthread_key_delete() is called. It is the responsibility of the application to free any
application storage or perform any cleanup actions for data structures related to the deleted key
or associated thread-specific data in any threads; this cleanup can be done either before or after
pthread_key_delete() is called. Any attempt to use key following the call to pthread_key_delete()
results in undefined behavior.

The pthread_key_delete() function shall be callable from within destructor functions. No
destructor functions shall be invoked by pthread_key_delete(). Any destructor function that may
have been associated with key shall no longer be called upon thread exit.

RETURN VALUE
If successful, the pthread_key_delete() function shall return zero; otherwise, an error number shall
be returned to indicate the error.

ERRORS
The pthread_key_delete() function may fail if:

[EINVAL] The key value is invalid.

The pthread_key_delete() function shall not return an error code of [EINTR].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
A thread-specific data key deletion function has been included in order to allow the resources
associated with an unused thread-specific data key to be freed. Unused thread-specific data keys
can arise, among other scenarios, when a dynamically loaded module that allocated a key is
unloaded.

Conforming applications are responsible for performing any cleanup actions needed for data
structures associated with the key to be deleted, including data referenced by thread-specific
data values. No such cleanup is done by pthread_key_delete(). In particular, destructor functions
are not called. There are several reasons for this division of responsibility:

1. The associated destructor functions used to free thread-specific data at thread exit time are
only guaranteed to work correctly when called in the thread that allocated the thread-
specific data. (Destructors themselves may utilize thread-specific data.) Thus, they cannot
be used to free thread-specific data in other threads at key deletion time. Attempting to
have them called by other threads at key deletion time would require other threads to be
asynchronously interrupted. But since interrupted threads could be in an arbitrary state,
including holding locks necessary for the destructor to run, this approach would fail. In
general, there is no safe mechanism whereby an implementation could free thread-specific
data at key deletion time.
2. Even if there were a means of safely freeing thread-specific data associated with keys to be deleted, doing so would require that implementations be able to enumerate the threads with non-NULL data and potentially keep them from creating more thread-specific data while the key deletion is occurring. This special case could cause extra synchronization in the normal case, which would otherwise be unnecessary.

For an application to know that it is safe to delete a key, it has to know that all the threads that might potentially ever use the key do not attempt to use it again. For example, it could know this if all the client threads have called a cleanup procedure declaring that they are through with the module that is being shut down, perhaps by setting a reference count to zero.

FUTURE DIRECTIONS
None.

SEE ALSO
pthread_key_create(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6
The pthread_key_delete() function is marked as part of the Threads option.
NAME
   pthread_kill — send a signal to a thread

SYNOPSIS
THR
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);

DESCRIPTION
   The pthread_kill() function shall request that a signal be delivered to the specified thread.

   As in kill(), if sig is zero, error checking shall be performed but no signal shall actually be sent.

RETURN VALUE
   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.

ERRORS
   The pthread_kill() function shall fail if:

   [ESRCH]  No thread could be found corresponding to that specified by the given thread
            ID.

   [EINVAL]  The value of the sig argument is an invalid or unsupported signal number.

   The pthread_kill() function shall not return an error code of [EINTR].

EXAMPLES
   None.

APPLICATION USAGE
   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.

   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.

RATIONALE
   None.

FUTURE DIRECTIONS
   None.

SEE ALSO
   kill(), pthread_self(), raise(), the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>

CHANGE HISTORY
   First released in Issue 5. Included for alignment with the POSIX Threads Extension.

   Issue 6
   The pthread_kill() function is marked as part of the Threads option.

   The APPLICATION USAGE section is added.
**NAME**

pthread_mutex_destroy, pthread_mutex_init — destroy and initialize a mutex

**SYNOPSIS**

```c
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
            const pthread_mutexattr_t *restrict attr);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

**DESCRIPTION**

The `pthread_mutex_destroy()` function shall destroy the mutex object referenced by `mutex`; the mutex object becomes, in effect, uninitialized. An implementation may cause `pthread_mutex_destroy()` to set the object referenced by `mutex` to an invalid value. A destroyed mutex object can be reinitialized using `pthread_mutex_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

The `pthread_mutex_init()` function shall initialize the mutex referenced by `mutex` with attributes specified by `attr`. If `attr` is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Only `mutex` itself may be used for performing synchronization. The result of referring to copies of `mutex` in calls to `pthread_mutex_lock()`, `pthread_mutex_trylock()`, `pthread_mutex_unlock()`, and `pthread_mutex_destroy()` is undefined.

Attempting to initialize an already initialized mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro `PTHREAD_MUTEX_INITIALIZER` can be used to initialize mutexes that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to `pthread_mutex_init()` with parameter `attr` specified as NULL, except that no error checks are performed.

**RETURN VALUE**

If successful, the `pthread_mutex_destroy()` and `pthread_mutex_init()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and shall cause an error return prior to modifying the state of the mutex specified by `mutex`.

**ERRORS**

The `pthread_mutex_destroy()` function may fail if:

- [EBUSY]: The implementation has detected an attempt to destroy the object referenced by `mutex` while it is locked or referenced (for example, while being used in a `pthread_cond_timedwait()` or `pthread_cond_wait()`) by another thread.

- [EINVAL]: The value specified by `mutex` is invalid.

The `pthread_mutex_init()` function shall fail if:

- [EAGAIN]: The system lacked the necessary resources (other than memory) to initialize another mutex.
The `pthread_mutex_init()` function may fail if:

- `[ENOMEM]` Insufficient memory exists to initialize the mutex.
- `[EPERM]` The caller does not have the privilege to perform the operation.
- `[EBUSY]` The implementation has detected an attempt to reinitialize the object referenced by `mutex`, a previously initialized, but not yet destroyed, mutex.
- `[EINVAL]` The value specified by `attr` is invalid.

These functions shall not return an error code of `[EINTR]`.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

**Alternate Implementations Possible**

This volume of IEEE Std 1003.1-2001 supports several alternative implementations of mutexes. An implementation may store the lock directly in the object of type `pthread_mutex_t`. Alternatively, an implementation may store the lock in the heap and merely store a pointer, handle, or unique ID in the mutex object. Either implementation has advantages or may be required on certain hardware configurations. So that portable code can be written that is invariant to this choice, this volume of IEEE Std 1003.1-2001 does not define assignment or equality for this type, and it uses the term “initialize” to reinforce the (more restrictive) notion that the lock may actually reside in the mutex object itself.

Note that this precludes an over-specification of the type of the mutex or condition variable and motivates the opaqueness of the type.

An implementation is permitted, but not required, to have `pthread_mutex_destroy()` store an illegal value into the mutex. This may help detect erroneous programs that try to lock (or otherwise reference) a mutex that has already been destroyed.

**Tradeoff Between Error Checks and Performance Supported**

Many of the error checks were made optional in order to let implementations trade off performance versus degree of error checking according to the needs of their specific applications and execution environment. As a general rule, errors or conditions caused by the system (such as insufficient memory) always need to be reported, but errors due to an erroneously coded application (such as failing to provide adequate synchronization to prevent a mutex from being deleted while in use) are made optional.

A wide range of implementations is thus made possible. For example, an implementation intended for application debugging may implement all of the error checks, but an implementation running a single, provably correct application under very tight performance constraints in an embedded computer might implement minimal checks. An implementation might even be provided in two versions, similar to the options that compilers provide: a full-checking, but slower version; and a limited-checking, but faster version. To forbid this optionality would be a disservice to users.

By carefully limiting the use of “undefined behavior” only to things that an erroneous (badly coded) application might do, and by defining that resource-not-available errors are mandatory, this volume of IEEE Std 1003.1-2001 ensures that a fully-conforming application is portable.
across the full range of implementations, while not forcing all implementations to add overhead
to check for numerous things that a correct program never does.

Why No Limits are Defined
Defining symbols for the maximum number of mutexes and condition variables was considered
but rejected because the number of these objects may change dynamically. Furthermore, many
implementations place these objects into application memory; thus, there is no explicit
maximum.

Static Initializers for Mutexes and Condition Variables
Providing for static initialization of statically allocated synchronization objects allows modules
with private static synchronization variables to avoid runtime initialization tests and overhead.
Furthermore, it simplifies the coding of self-initializing modules. Such modules are common in
C libraries, where for various reasons the design calls for self-initialization instead of requiring
an explicit module initialization function to be called. An example use of static initialization
follows.

Without static initialization, a self-initializing routine foo() might look as follows:

```c
static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
static pthread_mutex_t foo_mutex;

void foo_init()
{
    pthread_mutex_init(&foo_mutex, NULL);
}

void foo()
{
    pthread_once(&foo_once, foo_init);
    pthread_mutex_lock(&foo_mutex);
    /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

With static initialization, the same routine could be coded as follows:

```c
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;

void foo()
{
    pthread_mutex_lock(&foo_mutex);
    /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

Note that the static initialization both eliminates the need for the initialization test inside
pthread_once() and the fetch of &foo_mutex to learn the address to be passed to
pthread_mutex_lock() or pthread_mutex_unlock().

Thus, the C code written to initialize static objects is simpler on all systems and is also faster on a
large class of systems; those where the (entire) synchronization object can be stored in
application memory.

Yet the locking performance question is likely to be raised for machines that require mutexes to
be allocated out of special memory. Such machines actually have to have mutexes and possibly
condition variables contain pointers to the actual hardware locks. For static initialization to work
on such machines, `pthread_mutex_lock()` also has to test whether or not the pointer to the actual
lock has been allocated. If it has not, `pthread_mutex_lock()` has to initialize it before use. The
reservation of such resources can be made when the program is loaded, and hence return codes
have not been added to mutex locking and condition variable waiting to indicate failure to
complete initialization.

This runtime test in `pthread_mutex_lock()` would at first seem to be extra work; an extra test is
required to see whether the pointer has been initialized. On most machines this would actually
be implemented as a fetch of the pointer, testing the pointer against zero, and then using the
pointer if it has already been initialized. While the test might seem to add extra work, the extra
effort of testing a register is usually negligible since no extra memory references are actually
done. As more and more machines provide caches, the real expenses are memory references, not
instructions executed.

Alternatively, depending on the machine architecture, there are often ways to eliminate all
overhead in the most important case: on the lock operations that occur after the lock has been
initialized. This can be done by shifting more overhead to the less frequent operation:
initialization. Since out-of-line mutex allocation also means that an address has to be
dereferenced to find the actual lock, one technique that is widely applicable is to have static
initialization store a bogus value for that address; in particular, an address that causes a machine
fault to occur. When such a fault occurs upon the first attempt to lock such a mutex, validity
checks can be done, and then the correct address for the actual lock can be filled in. Subsequent
lock operations incur no extra overhead since they do not “fault”. This is merely one technique
that can be used to support static initialization, while not adversely affecting the performance of
lock acquisition. No doubt there are other techniques that are highly machine-dependent.

The locking overhead for machines doing out-of-line mutex allocation is thus similar for
modules being implicitly initialized, where it is improved for those doing mutex allocation
totally inline. The inline case is thus made much faster, and the out-of-line case is not
significantly worse.

Besides the issue of locking performance for such machines, a concern is raised that it is possible
that threads would serialize contending for initialization locks when attempting to finish
initializing statically allocated mutexes. (Such finishing would typically involve taking an
internal lock, allocating a structure, storing a pointer to the structure in the mutex, and releasing
the internal lock.) First, many implementations would reduce such serialization by hashing on
the mutex address. Second, such serialization can only occur a bounded number of times. In
particular, it can happen at most as many times as there are statically allocated synchronization
objects. Dynamically allocated objects would still be initialized via `pthread_mutex_init()` or
`pthread_cond_init()`.

Finally, if none of the above optimization techniques for out-of-line allocation yields sufficient
performance for an application on some implementation, the application can avoid static
initialization altogether by explicitly initializing all synchronization objects with the
corresponding `pthread_*_init()` functions, which are supported by all implementations. An
implementation can also document the tradeoffs and advise which initialization technique is
more efficient for that particular implementation.
Destroying Mutexes

A mutex can be destroyed immediately after it is unlocked. For example, consider the following code:

```c
struct obj {
  pthread_mutex_t om;
  int refcnt;
  ...
};

obj_done(struct obj *op) {
  pthread_mutex_lock(&op->om);
  if (--op->refcnt == 0) {
    pthread_mutex_unlock(&op->om);
    (A) pthread_mutex_destroy(&op->om);
    (B) free(op);
  } else
    (C) pthread_mutex_unlock(&op->om);
}
```

In this case `obj` is reference counted and `obj_done()` is called whenever a reference to the object is dropped. Implementations are required to allow an object to be destroyed and freed and potentially unmapped (for example, lines A and B) immediately after the object is unlocked (line C).

FUTURE DIRECTIONS

None.

SEE ALSO

`pthread_mutex_getprioceiling()`, `pthread_mutex_lock()`, `pthread_mutex_timedlock()`, `pthread_mutexattr_getpshared()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The `pthread_mutex_destroy()` and `pthread_mutex_init()` functions are marked as part of the Threads option.

The `pthread_mutex_timedlock()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

The `restrict` keyword is added to the `pthread_mutex_init()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME

pthread_mutex_getprioceiling, pthread_mutex_setprioceiling — get and set the priority ceiling of a mutex (REALTIME THREADS)

SYNOPSIS

```c
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
                                 int *restrict prioceiling);

int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
                                 int prioceiling, int *restrict old_ceiling);
```

DESCRIPTION

The `pthread_mutex_getprioceiling()` function shall return the current priority ceiling of the mutex.

The `pthread_mutex_setprioceiling()` function shall either lock the mutex if it is unlocked, or block until it can successfully lock the mutex, then it shall change the mutex’s priority ceiling and release the mutex. When the change is successful, the previous value of the priority ceiling shall be returned in `old_ceiling`. The process of locking the mutex need not adhere to the priority protect protocol.

If the `pthread_mutex_setprioceiling()` function fails, the mutex priority ceiling shall not be changed.

RETURN VALUE

If successful, the `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions may fail if:

- [EINVAL] The priority requested by `prioceiling` is out of range.
- [EINVAL] The value specified by `mutex` does not refer to a currently existing mutex.
- [EPERM] The caller does not have the privilege to perform the operation.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`pthread_mutex_destroy()`, `pthread_mutex_lock()`, `pthread_mutex_timedlock()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.
The `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions are marked as part of the Threads and Thread Priority Protection options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Priority Protection option.

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.

The `pthread_mutex_timedlock()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

The `restrict` keyword is added to the `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` prototypes for alignment with the ISO/IEC 9899:1999 standard.
NAME
pthread_mutex_init — initialize a mutex

SYNOPSIS
THR

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
const pthread_mutexattr_t *restrict attr);
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

DESCRIPTION
Refer to `pthread_mutex_destroy()`.
NAME
pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

SYNOPSIS

```c
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

The mutex object referenced by `mutex` shall be locked by calling `pthread_mutex_lock()`. If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by `mutex` in the locked state with the calling thread as its owner.

If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection shall not be provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results.

If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking shall be provided. If a thread attempts to relock a mutex that it has already locked, an error shall be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error shall be returned.

If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex shall maintain the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count shall be set to one. Every time a thread relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error shall be returned.

If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior.

The `pthread_mutex_trylock()` function shall be equivalent to `pthread_mutex_lock()`, except that if the mutex object referenced by `mutex` is currently locked (by any thread, including the current thread), the call shall return immediately. If the mutex type is PTHREAD_MUTEX_RECURSIVE and the mutex is currently owned by the calling thread, the mutex lock count shall be incremented by one and the `pthread_mutex_trylock()` function shall immediately return success.

The `pthread_mutex_unlock()` function shall release the mutex object referenced by `mutex`. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by `mutex` when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

(In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex shall become available when the count reaches zero and the calling thread no longer has any locks on this mutex.)

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread shall resume waiting for the mutex as if it was not interrupted.
RETURN VALUE
If successful, the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The `pthread_mutex_trylock()` function shall return zero if a lock on the mutex object referenced by `mutex` is acquired. Otherwise, an error number is returned to indicate the error.

ERRORS
The `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions shall fail if:

- [EINVAL] The `mutex` was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT and the calling thread’s priority is higher than the mutex’s current priority ceiling.

The `pthread_mutex_trylock()` function shall fail if:

- [EBUSY] The mutex could not be acquired because it was already locked.

The `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` functions may fail if:

- [EINVAL] The value specified by `mutex` does not refer to an initialized mutex object.

- [EAGAIN] The mutex could not be acquired because the maximum number of recursive locks for `mutex` has been exceeded.

- [EDEADLK] The current thread already owns the mutex.

- [EPERM] The current thread does not own the mutex.

These functions shall not return an error code of [EINVAL].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
Mutex objects are intended to serve as a low-level primitive from which other thread synchronization functions can be built. As such, the implementation of mutexes should be as efficient as possible, and this has ramifications on the features available at the interface.

The mutex functions and the particular default settings of the mutex attributes have been motivated by the desire to not preclude fast, inlined implementations of mutex locking and unlocking.

For example, deadlocking on a double-lock is explicitly allowed behavior in order to avoid requiring more overhead in the basic mechanism than is absolutely necessary. (More “friendly” mutexes that detect deadlock or that allow multiple locking by the same thread are easily constructed by the user via the other mechanisms provided. For example, `pthread_self()` can be used to record mutex ownership.) Implementations might also choose to provide such extended features as options via special mutex attributes.

Since most attributes only need to be checked when a thread is going to be blocked, the use of attributes does not slow the (common) mutex-locking case.
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 `mutex_tryunlock` operation.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`pthread_mutex_destroy()`, `pthread_mutex_timedlock()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

**CHANGE HISTORY**

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

**Issue 6**

The `pthread_mutex_lock()`, `pthread_mutex_trylock()`, and `pthread_mutex_unlock()` functions are marked as part of the Threads option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The behavior when attempting to relock a mutex is defined.

The `pthread_mutex_timedlock()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
NAME
pthread_mutex_setprioceiling — change the priority ceiling of a mutex (REALTIME THREADS)

SYNOPSIS
#include <pthread.h>

int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex, int prioceiling, int *restrict old_ceiling);

DESCRIPTION
Refer to pthread_mutex_getprioceiling().
NAME

pthread_mutex_timedlock — lock a mutex (ADVANCED REALTIME)

SYNOPSIS

```c
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
const struct timespec *restrict abs_timeout);
```

DESCRIPTION

The `pthread_mutex_timedlock()` function shall lock the mutex object referenced by `mutex`. If the mutex is already locked, the calling thread shall block until the mutex becomes available as in the `pthread_mutex_lock()` function. If the mutex cannot be locked without waiting for another thread to unlock the mutex, 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.

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 circumstance shall the function fail with a timeout if the mutex can be locked immediately. The validity of the `abs_timeout` parameter need not be checked if the mutex can be locked immediately.

As a consequence of the priority inheritance rules (for mutexes initialized with the PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the priority of the owner of the mutex shall be adjusted as necessary to reflect the fact that this thread is no longer among the threads waiting for the mutex.

RETURN VALUE

If successful, the `pthread_mutex_timedlock()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The `pthread_mutex_timedlock()` function shall fail if:

- **[EINVAL]** The mutex was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT and the calling thread’s priority is higher than the mutex' current priority ceiling.

- **[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.

- **[ETIMEDOUT]** The mutex could not be locked before the specified timeout expired.

The `pthread_mutex_timedlock()` function may fail if:

- **[EINVAL]** The value specified by `mutex` does not refer to an initialized mutex object.
The mutex could not be acquired because the maximum number of recursive locks for mutex has been exceeded.

The current thread already owns the mutex.

This function shall not return an error code of [EINTR].

**EXAMPLES**

None.

**APPLICATION USAGE**

The `pthread_mutex_timedlock()` function is part of the Threads and Timeouts options and need not be provided on all implementations.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`pthread_mutex_destroy()`, `pthread_mutex_lock()`, `pthread_mutex_trylock()`, `time()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`, `<time.h>`

**CHANGE HISTORY**

NAME
pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

SYNOPSIS
THR

#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

DESCRIPTION
Refer to pthread_mutex_lock().
NAME

pthread_mutexattr_destroy, pthread_mutexattr_init — destroy and initialize the mutex attributes object

SYNOPSIS

THR

```c
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

DESCRIPTION

The `pthread_mutexattr_destroy()` function shall destroy a mutex attributes object; the object becomes, in effect, uninitialized. An implementation may cause `pthread_mutexattr_destroy()` to set the object referenced by `attr` to an invalid value. A destroyed `attr` attributes object can be reinitialized using `pthread_mutexattr_init()``.; the results of otherwise referencing the object after it has been destroyed are undefined.

The `pthread_mutexattr_init()` function shall initialize a mutex attributes object `attr` with the default value for all of the attributes defined by the implementation.

Results are undefined if `pthread_mutexattr_init()` is called specifying an already initialized `attr` attributes object.

After a mutex attributes object has been used to initialize one or more mutexes, any function affecting the attributes object (including destruction) shall not affect any previously initialized mutexes.

RETURN VALUE

Upon successful completion, `pthread_mutexattr_destroy()` and `pthread_mutexattr_init()` shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The `pthread_mutexattr_destroy()` function may fail if:

- [EINVAL] The value specified by `attr` is invalid.

The `pthread_mutexattr_init()` function shall fail if:

- [ENOMEM] Insufficient memory exists to initialize the mutex attributes object.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

See `pthread_attr_init()` for a general explanation of attributes. Attributes objects allow implementations to experiment with useful extensions and permit extension of this volume of IEEE Std 1003.1-2001 without changing the existing functions. Thus, they provide for future extensibility of this volume of IEEE Std 1003.1-2001 and reduce the temptation to standardize prematurely on semantics that are not yet widely implemented or understood.

Examples of possible additional mutex attributes that have been discussed are `spin_only`, `limited_spin`, `no_spin`, `recursive`, and `metered`. (To explain what the latter attributes might mean: recursive mutexes would allow for multiple re-locking by the current owner; metered mutexes would transparently keep records of queue length, wait time, and so on.) Since there is not yet
wide agreement on the usefulness of these resulting from shared implementation and usage experience, they are not yet specified in this volume of IEEE Std 1003.1-2001. Mutex attributes objects, however, make it possible to test out these concepts for possible standardization at a later time.

**Mutex Attributes and Performance**

Care has been taken to ensure that the default values of the mutex attributes have been defined such that mutexes initialized with the defaults have simple enough semantics so that the locking and unlocking can be done with the equivalent of a test-and-set instruction (plus possibly a few other basic instructions).

There is at least one implementation method that can be used to reduce the cost of testing at lock-time if a mutex has non-default attributes. One such method that an implementation can employ (and this can be made fully transparent to fully conforming POSIX applications) is to secretly pre-lock any mutexes that are initialized to non-default attributes. Any later attempt to lock such a mutex causes the implementation to branch to the "slow path" as if the mutex were unavailable; then, on the slow path, the implementation can do the "real work" to lock a non-default mutex. The underlying unlock operation is more complicated since the implementation never really wants to release the pre-lock on this kind of mutex. This illustrates that, depending on the hardware, there may be certain optimizations that can be used so that whatever mutex attributes are considered "most frequently used" can be processed most efficiently.

**Process Shared Memory and Synchronization**

The existence of memory mapping functions in this volume of IEEE Std 1003.1-2001 leads to the possibility that an application may allocate the synchronization objects from this section in memory that is accessed by multiple processes (and therefore, by threads of multiple processes).

In order to permit such usage, while at the same time keeping the usual case (that is, usage within a single process) efficient, a *process-shared* option has been defined.

If an implementation supports the _POSIX_THREAD_PROCESS_SHARED option, then the *process-shared* attribute can be used to indicate that mutexes or condition variables may be accessed by threads of multiple processes.

The default setting of PTHREAD_PROCESS_PRIVATE has been chosen for the *process-shared* attribute so that the most efficient forms of these synchronization objects are created by default.

Synchronization variables that are initialized with the PTHREAD_PROCESS_PRIVATE *process-shared* attribute may only be operated on by threads in the process that initialized them. Synchronization variables that are initialized with the PTHREAD_PROCESS_SHARED *process-shared* attribute may be operated on by any thread in any process that has access to it. In particular, these processes may exist beyond the lifetime of the initializing process. For example, the following code implements a simple counting semaphore in a mapped file that may be used by many processes.

```c
/* sem.h */
struct semaphore {
    pthread_mutex_t lock;
    pthread_cond_t nonzero;
    unsigned count;
};
typedef struct semaphore semaphore_t;
semaphore_t *semaphore_create(char *semaphore_name);
semaphore_t *semaphore_open(char *semaphore_name);
```
void semaphore_post(semaphore_t *semap);
void semaphore_wait(semaphore_t *semap);
void semaphore_close(semaphore_t *semap);

/* sem.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include "sem.h"

semaphore_t *
semaphore_create(char *semaphore_name)
{
    int fd;
    semaphore_t *semap;
    pthread_mutexattr_t psharedm;
    pthread_condattr_t psharedc;
    fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
    if (fd < 0)
        return (NULL);
    (void) ftruncate(fd, sizeof(semaphore_t));
    (void) pthread_mutexattr_init(&psharedm);
    (void) pthread_mutexattr_setpshared(&psharedm,
        PTHREAD_PROCESS_SHARED);
    (void) pthread_condattr_init(&psharedc);
    (void) pthread_condattr_setpshared(&psharedc,
        PTHREAD_PROCESS_SHARED);
    semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
        PROT_READ | PROT_WRITE, MAP_SHARED,
        fd, 0);
    close (fd);
    (void) pthread_mutex_init(&semap->lock, &psharedm);
    (void) pthread_cond_init(&semap->nonzero, &psharedc);
    semap->count = 0;
    return (semap);
}

semaphore_t *
semaphore_open(char *semaphore_name)
{
    int fd;
    semaphore_t *semap;
    fd = open(semaphore_name, O_RDWR, 0666);
    if (fd < 0)
        return (NULL);
    semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
        PROT_READ | PROT_WRITE, MAP_SHARED,
        fd, 0);
    close (fd);
    (void) pthread_mutex_init(&semap->lock, &psharedm);
    (void) pthread_cond_init(&semap->nonzero, &psharedc);
    semap->count = 0;
    return (semap);
}
The following code is for three separate processes that create, post, and wait on a semaphore in the file /tmp/semaphore. Once the file is created, the post and wait programs increment and decrement the counting semaphore (waiting and waking as required) even though they did not initialize the semaphore.

The code is as follows:

```c
#include "pthread.h"
#include "sem.h"

int main()
{
    semaphore_t *semap;
    semap = semaphore_create("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_close(semap);
    return (0);
}
```
```c
#include "sem.h"
#include "semaphore.h"

int main()
{
    semaphore_t *semap;
    semap = semaphore_open("/tmp/semaphe");
    if (semap == NULL)
        exit(1);
    semaphore_wait(semap);
    semaphore_close(semap);
    return (0);
}
```

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`pthread_cond_destroy()`, `pthread_create()`, `pthread_mutex_destroy()`, `pthread_mutexattr_destroy()`, the

**CHANGE HISTORY**

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

**Issue 6**

The `pthread_mutexattr_destroy()` and `pthread_mutexattr_init()` functions are marked as part of the

Threads option.

IEEE PASC Interpretation 1003.1c #27 is applied, updating the ERRORS section.
NAME

pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling — get and set the
prioceiling attribute of the mutex attributes object (REALTIME THREADS)

SYNOPSIS

THR TYP
#include <pthread.h>

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t * restrict attr, int *restrict prioceiling);
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);

DESCRIPTION

The pthread_mutexattr_getprioceiling() and pthread_mutexattr_setprioceiling() functions,
respectively, shall get and set the priority ceiling attribute of a mutex attributes object pointed to
by attr which was previously created by the function pthread_mutexattr_init().

The prioceiling attribute contains the priority ceiling of initialized mutexes. The values of
prioceiling are within the maximum range of priorities defined by SCHED_FIFO.

The prioceiling attribute defines the priority ceiling of initialized mutexes, which is the minimum
priority level at which the critical section guarded by the mutex is executed. In order to avoid
priority inversion, the priority ceiling of the mutex shall be set to a priority higher than or equal
to the highest priority of all the threads that may lock that mutex. The values of prioceiling are
within the maximum range of priorities defined under the SCHED_FIFO scheduling policy.

RETURN VALUE

Upon successful completion, the pthread_mutexattr_getprioceiling() and
pthread_mutexattr_setprioceiling() functions shall return zero; otherwise, an error number shall be
returned to indicate the error.

ERRORS

The pthread_mutexattr_getprioceiling() and pthread_mutexattr_setprioceiling() functions may fail if:

[EINVAL] The value specified by attr or prioceiling is invalid.
[EPERM] The caller does not have the privilege to perform the operation.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy(), pthread_create(), pthread_mutex_destroy(), the Base Definitions volume of
IEEE Std 1003.1-2001, <pthread.h>
**CHANGE HISTORY**

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

**Issue 6**

The `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions are marked as part of the Threads and Thread Priority Protection options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Priority Protection option.

The [ENOTSUP] error condition has been removed since these functions do not have a protocol argument.

The `restrict` keyword is added to the `pthread_mutexattr_getprioceiling()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME

pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol — get and set the protocol attribute of the mutex attributes object (REALTIME THREADS)

SYNOPSIS

THR

#include <pthread.h>

TPP|TPI

int pthread_mutexattr_getprotocol(const pthread_mutexattr_t * restrict attr, int *restrict protocol);

int pthread_mutexattr_setprotocol(pthread_mutexattr_t * attr, int protocol);

DESCRIPTION

The pthread_mutexattr_getprotocol() and pthread_mutexattr_setprotocol() functions, respectively, shall get and set the protocol attribute of a mutex attributes object pointed to by attr which was previously created by the function pthread_mutexattr_init().

The protocol attribute defines the protocol to be followed in utilizing mutexes. The value of protocol may be one of:

PTHREAD_PRIO_NONE

PTHREAD_PRIO_INHERIT

PTHREAD_PRIO_PROTECT

which are defined in the <pthread.h> header.

When a thread owns a mutex with the PTHREAD_PRIO_NONE protocol attribute, its priority and scheduling shall not be affected by its mutex ownership.

When a thread is blocking higher priority threads because of owning one or more mutexes with the PTHREAD_PRIO_INHERIT protocol attribute, it shall execute at the higher of its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol.

When a thread owns one or more mutexes initialized with the PTHREAD_PRIO_PROTECT protocol, it shall execute at the higher of its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these mutexes or not.

While a thread is holding a mutex which has been initialized with the PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT protocol attributes, it shall not be subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed, such as by a call to sched_setparam(). Likewise, when a thread unlocks a mutex that has been initialized with the PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT protocol attributes, it shall not be subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed.

If a thread simultaneously owns several mutexes initialized with different protocols, it shall execute at the highest of the priorities that it would have obtained by each of these protocols.

When a thread makes a call to pthread_mutex_lock(), the mutex was initialized with the protocol attribute having the value PTHREAD_PRIO_INHERIT, when the calling thread is blocked because the mutex is owned by another thread, that owner thread shall inherit the priority level of the calling thread as long as it continues to own the mutex. The implementation shall update its execution priority to the maximum of its assigned priority and all its inherited priorities. Furthermore, if this owner thread itself becomes blocked on another mutex, the same priority
inherence effect shall be propagated to this other owner thread, in a recursive manner.

RETURN VALUE
Upon successful completion, the `pthread_mutexattr_getprotocol()` and
`pthread_mutexattr_setprotocol()` functions shall return zero; otherwise, an error number shall be
returned to indicate the error.

ERRORS
The `pthread_mutexattr_setprotocol()` function shall fail if:
- [ENOTSUP] The value specified by `protocol` is an unsupported value.

The `pthread_mutexattr_getprotocol()` and `pthread_mutexattr_setprotocol()` functions may fail if:
- [EINVAL] The value specified by `attr` or `protocol` is invalid.
- [EPERM] The caller does not have the privilege to perform the operation.

These functions shall not return an error code of [EINVAL].

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
`pthread_cond_destroy()`, `pthread_create()`, `pthread_mutex_destroy()`, the Base Definitions volume of
IEEE Std 1003.1-2001, `<pthread.h>`

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.
Marked as part of the Realtime Threads Feature Group.

Issue 6
The `pthread_mutexattr_getprotocol()` and `pthread_mutexattr_setprotocol()` functions are marked as
part of the Threads option and either the Thread Priority Protection or Thread Priority
Inheritance options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an
implementation does not support the Thread Priority Protection or Thread Priority Inheritance
options.

The `restrict` keyword is added to the `pthread_mutexattr_getprotocol()` prototype for alignment
NAME
pthread_mutexattr_getpshared, pthread_mutexattr_setpshared — get and set the process-shared attribute

SYNOPSIS
#include <pthread.h>

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict attr, int *restrict pshared);

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);

DESCRIPTION
The pthread_mutexattr_getpshared() function shall obtain the value of the process-shared attribute from the attributes object referenced by attr. The pthread_mutexattr_setpshared() function shall 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 mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes. If the process-shared attribute is PTHREAD_PROCESS_PRIVATE, the mutex shall only be operated upon by threads created within the same process as the thread that initialized the mutex; if threads of differing processes attempt to operate on such a mutex, the behavior is undefined. The default value of the attribute shall be PTHREAD_PROCESS_PRIVATE.

RETURN VALUE
Upon successful completion, pthread_mutexattr_setpshared() shall return zero; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, pthread_mutexattr_getpshared() 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.

ERRORS
The pthread_mutexattr_getpshared() and pthread_mutexattr_setpshared() functions may fail if:

[EINVAL] The value specified by attr is invalid.

The pthread_mutexattr_setpshared() function may fail if:

[EINVAL] The new value specified for the attribute is outside the range of legal values for that attribute.

These functions shall not return an error code of [EINVAL].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

1096

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
SEE ALSO

pthread_cond_destroy(), pthread_create(), pthread_mutex_destroy(), pthread_mutexattr_destroy(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

The pthread_mutexattr_getpshared() and pthread_mutexattr_setpshared() functions are marked as part of the Threads and Thread Process-Shared Synchronization options.

The restrict keyword is added to the pthread_mutexattr_getpshared() prototype for alignment with the ISO/IEC 9899: 1999 standard.
NAME
pthread_mutexattr_gettype, pthread_mutexattr_settype — get and set the mutex type attribute

SYNOPSIS
XSI
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
   int *restrict type);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);

DESCRIPTION
The pthread_mutexattr_gettype() and pthread_mutexattr_settype() functions, respectively, shall get
and set the mutex type attribute. This attribute is set in the type parameter to these functions. The
default value of the type attribute is PTHREAD_MUTEX_DEFAULT.

The type of mutex is contained in the type attribute of the mutex attributes. Valid mutex types
include:

PTHREAD_MUTEX_NORMAL
This type of mutex does not detect deadlock. A thread attempting to relock this mutex
without first unlocking it shall deadlock. Attempting to unlock a mutex locked by a
different thread results in undefined behavior. Attempting to unlock an unlocked mutex
results in undefined behavior.

PTHREAD_MUTEX_ERRORCHECK
This type of mutex provides error checking. A thread attempting to relock this mutex
without first unlocking it shall return with an error. A thread attempting to unlock a mutex
which another thread has locked shall return with an error. A thread attempting to unlock
an unlocked mutex shall return with an error.

PTHREAD_MUTEX_RECURSIVE
A thread attempting to relock this mutex without first unlocking it shall succeed in locking
the mutex. The relocking deadlock which can occur with mutexes of type
PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this
mutex shall require the same number of unlocks to release the mutex before another thread
can acquire the mutex. A thread attempting to unlock a mutex which another thread has
locked shall return with an error. A thread attempting to unlock an unlocked mutex shall
return with an error.

PTHREAD_MUTEX_DEFAULT
Attempting to recursively lock a mutex of this type results in undefined behavior.
Attempting to unlock a mutex of this type which was not locked by the calling thread
results in undefined behavior. Attempting to unlock a mutex of this type which is not
locked results in undefined behavior. An implementation may map this mutex to one of the
other mutex types.

RETURN VALUE
Upon successful completion, the pthread_mutexattr_gettype() function shall return zero and store
the value of the type attribute of attr into the object referenced by the type parameter. Otherwise,
an error shall be returned to indicate the error.

If successful, the pthread_mutexattr_settype() function shall return zero; otherwise, an error
number shall be returned to indicate the error.
The `pthread_mutexattr_settype()` function shall fail if:

- `[EINVAL]` The value `type` is invalid.

The `pthread_mutexattr_gettype()` and `pthread_mutexattr_settype()` functions may fail if:

- `[EINVAL]` The value specified by `attr` is invalid.

These functions shall not return an error code of `[EINVAL]`.

**EXAMPLES**

None.

**APPLICATION USAGE**

It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with condition variables because the implicit unlock performed for a `pthread_cond_timedwait()` or `pthread_cond_wait()` may not actually release the mutex (if it had been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`pthread_cond_timedwait()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

**CHANGE HISTORY**

First released in Issue 5.

**Issue 6**

The Open Group Corrigendum U033/3 is applied. The SYNOPSIS for `pthread_mutexattr_gettype()` is updated so that the first argument is of type `const pthread_mutexattr_t *`. The `restrict` keyword is added to the `pthread_mutexattr_gettype()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
pthread_mutexattr_init — initialize the mutex attributes object

SYNOPSIS
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

DESCRIPTION
Refer to pthread_mutexattr_destroy().
NAME

pthread_mutexattr_setprioceiling — set the prioceiling attribute of the mutex attributes object

(REALTIME THREADS)

SYNOPSIS

```c
#include <pthread.h>

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
int prioceiling);
```

DESCRIPTION

Refer to `pthread_mutexattr_getprioceiling()`.
NAME
pthread_mutexattr_setprotocol — set the protocol attribute of the mutex attributes object
(REALTIME THREADS)

SYNOPSIS
THR
#include <pthread.h>

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
int protocol);

DESCRIPTION
Refer to pthread_mutexattr_getprotocol().
NAME

pthread_mutexattr_setpshared — set the process-shared attribute

SYNOPSIS

```c
#include <pthread.h>

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

DESCRIPTION

Refer to `pthread_mutexattr_getpshared()`.
NAME
pthread_mutexattr_settype — set the mutex type attribute

SYNOPSIS
#include <pthread.h>

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);

DESCRIPTION
Refer to pthread_mutexattr_gettype().
NAME

pthread_once — dynamic package initialization

SYNOPSIS

```c
#include <pthread.h>

int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));

pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

DESCRIPTION

The first call to `pthread_once()` by any thread in a process, with a given `once_control`, shall call the `init_routine` with no arguments. Subsequent calls of `pthread_once()` with the same `once_control` shall not call the `init_routine`. On return from `pthread_once()`, `init_routine` shall have completed. The `once_control` parameter shall determine whether the associated initialization routine has been called.

The `pthread_once()` function is not a cancelation point. However, if `init_routine` is a cancelation point and is canceled, the effect on `once_control` shall be as if `pthread_once()` was never called.

The constant PTHREAD_ONCE_INIT is defined in the `<pthread.h>` header.

The behavior of `pthread_once()` is undefined if `once_control` has automatic storage duration or is not initialized by PTHREAD_ONCE_INIT.

RETURN VALUE

Upon successful completion, `pthread_once()` shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The `pthread_once()` function may fail if:

- [EINVAL] If either `once_control` or `init_routine` is invalid.
- [EINVAL] The `pthread_once()` function shall not return an error code of [EINVAL].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Some C libraries are designed for dynamic initialization. That is, the global initialization for the library is performed when the first procedure in the library is called. In a single-threaded program, this is normally implemented using a static variable whose value is checked on entry to a routine, as follows:

```c
static int random_is_initialized = 0;
extern int initialize_random();

int random_function()
{
    if (random_is_initialized == 0) {
        initialize_random();
        random_is_initialized = 1;
    }
    ...
    /* Operations performed after initialization. */
}```
To keep the same structure in a multi-threaded program, a new primitive is needed. Otherwise, library initialization has to be accomplished by an explicit call to a library-exported initialization function prior to any use of the library.

For dynamic library initialization in a multi-threaded process, a simple initialization flag is not sufficient; the flag needs to be protected against modification by multiple threads simultaneously calling into the library. Protecting the flag requires the use of a mutex; however, mutexes have to be initialized before they are used. Ensuring that the mutex is only initialized once requires a recursive solution to this problem.

The use of `pthread_once()` not only supplies an implementation-guaranteed means of dynamic initialization, it provides an aid to the reliable construction of multi-threaded and realtime systems. The preceding example then becomes:

```c
#include <pthread.h>
static pthread_once_t random_is_initialized = PTHREAD_ONCE_INIT;
extern int initialize_random();

int random_function()
{
    (void) pthread_once(&random_is_initialized, initialize_random);
    ... /* Operations performed after initialization. */
}
```

Note that a `pthread_once_t` cannot be an array because some compilers do not accept the construct `&<array_name>`. 

### FUTURE DIRECTIONS
None.

### SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

### CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Threads Extension.

**Issue 6**

The `pthread_once()` function is marked as part of the Threads option.

The [EINVAL] error is added as a may fail case for if either argument is invalid.
NAME

pthread_rwlock_destroy, pthread_rwlock_init — destroy and initialize a read-write lock object

SYNOPSIS

THR

#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
 const pthread_rwlockattr_t *restrict attr);

DESCRIPTION

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.

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.

If the pthread_rwlock_init() function fails, rwlock shall not be initialized and the contents of rwlock
are undefined.

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.

RETURN VALUE

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.

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.

ERRORS

The pthread_rwlock_destroy() function may fail if:

[EBUSY] The implementation has detected an attempt to destroy the object referenced
by rwlock while it is locked.

[EINVAL] The value specified by rwlock is invalid.

The pthread_rwlock_init() function shall fail if:

[EAGAIN] The system lacked the necessary resources (other than memory) to initialize
another read-write lock.

[ENOMEM] Insufficient memory exists to initialize the read-write lock.

[EPERM] The caller does not have the privilege to perform the operation.
The `pthread_rwlock_init()` function may fail if:

- `EBUSY` The implementation has detected an attempt to reinitialize the object referenced by `rwlock`, a previously initialized but not yet destroyed read-write lock.
- `EINVAL` The value specified by `attr` is invalid.

These functions shall not return an error code of `[EINVAL]`.

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`pthread_rwlock_rdlock()`, `pthread_rwlock_timedrdlock()`, `pthread_rwlock_timedwrlock()`,
`pthread_rwlock_tryrdlock()`, `pthread_rwlock_trywrlock()`, `pthread_rwlock_unlock()`,
`pthread_rwlock_wrlock()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

**CHANGE HISTORY**
First released in Issue 5.

**Issue 6**
The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 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). The initializer macro is also deleted from the SYNOPSIS.
- The DESCRIPTION is updated as follows:
  - It explicitly notes allocation of resources upon initialization of a read-write lock object.
  - A paragraph is added specifying that copies of read-write lock objects may not be used.
- An `EINVAL` error is added to the ERRORS section for `pthread_rwlock_init()`, indicating that the `rwlock` value is invalid.
- The SEE ALSO section is updated.

The `restrict` keyword is added to the `pthread_rwlock_init()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
pthread_rwlock_rdlock, pthread_rwlock_tryrdlock — lock a read-write lock object for reading

SYNOPSIS
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

DESCRIPTION
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.

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.

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.

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.

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).

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.

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.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

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.

RETURN VALUE
If successful, the pthread_rwlock_rdlock() function shall return zero; otherwise, an error number shall be returned to indicate the error.

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.
The `pthread_rwlock_rdlock()` function shall fail if:

- **[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.

The `pthread_rwlock_rdlock()` and `pthread_rwlock_tryrdlock()` functions may fail if:

- **[EINVAL]** The value specified by `rwlock` does not refer to an initialized read-write lock object.
- **[EAGAIN]** The read lock could not be acquired because the maximum number of read locks for `rwlock` has been exceeded.

The `pthread_rwlock_rdlock()` function may fail if:

- **[EDEADLK]** The current thread already owns the read-write lock for writing.

These functions shall not return an error code of **[EINTR]**.

**EXAMPLES**

None.

**APPLICATION USAGE**

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.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`pthread_rwlock_destroy()`, `pthread_rwlock_timedrdlock()`, `pthread_rwlock_timedwrlock()`, `pthread_rwlock_trywrlock()`, `pthread_rwlock_unlock()`, `pthread_rwlock_wrlock()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`

**CHANGE HISTORY**

First released in Issue 5.

**Issue 6**

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 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).

- The DESCRIPTION is updated as follows:
  - Conditions under which writers have precedence over readers are specified.
  - Failure of `pthread_rwlock_tryrdlock()` is clarified.
  - A paragraph on the maximum number of read locks is added.

- In the ERRORS sections, [EBUSY] is modified to take into account write priority, and [EDEADLK] is deleted as a `pthread_rwlock_tryrdlock()` error.

- The SEE ALSO section is updated.
NAME

pthread_rwlock_timedrdlock — lock a read-write lock for reading

SYNOPSIS

#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
const struct timespec *restrict abs_timeout);

DESCRIPTION

The `pthread_rwlock_timedrdlock()` function shall apply a read lock to the read-write lock referenced by `rwlock` as in the `pthread_rwlock_rdlock()` 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.

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.

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_timedrdlock()`, upon return from the signal handler the thread shall resume waiting for the lock as if it was not interrupted.

The calling thread may deadlock if at the time the call is made it holds a write lock on `rwlock`. The results are undefined if this function is called with an uninitialized read-write lock.

RETURN VALUE

The `pthread_rwlock_timedrdlock()` 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.

ERRORS

The `pthread_rwlock_timedrdlock()` function shall fail if:

- `[ETIMEDOUT]` The lock could not be acquired before the specified timeout expired.
- `[EAGAIN]` The read lock could not be acquired because the maximum number of read locks for lock would be exceeded.
- `[EDEADLK]` The calling thread already holds a write lock on `rwlock`.
- `[EINVAL]` The value specified by `rwlock` does not refer to an initialized read-write lock object, or the `abs_timeout` nanosecond value is less than zero or greater than or equal to 1 000 million.
- `[EINTR]` This function shall not return an error code of `[EINTR]`. 
EXAMPLES
None.

APPLICATION USAGE
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.

The `pthread_rwlock_timedrdlock()` function is part of the Threads and Timeouts options and need not be provided on all implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
`pthread_rwlock_destroy()`, `pthread_rwlock_rdlock()`, `pthread_rwlock_timedwrlock()`,
`pthread_rwlock_tryrdlock()`, `pthread_rwlock_trywrlock()`, `pthread_rwlock_unlock()`,
`pthread_rwlock_wrlock()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`, `<time.h>`

CHANGE HISTORY
NAME

pthread_rwlock_timedwrlock — lock a read-write lock for writing

SYNOPSIS

```c
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                              const struct timespec *restrict abs_timeout);
```

DESCRIPTION

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.

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.

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.

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.

RETURN VALUE

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.

ERRORS

The `pthread_rwlock_timedwrlock()` function shall fail if:

- `[ETIMEDOUT]` The lock could not be acquired before the specified timeout expired.

The `pthread_rwlock_timedwrlock()` function may fail if:

- `[EDEADLK]` The calling thread already holds the `rwlock`.
- `[EINVAL]` The value specified by `rwlock` does not refer to an initialized read-write lock object, or the `abs_timeout` nanosecond value is less than zero or greater than or equal to 1 000 million.

This function shall not return an error code of `[EINTR]`. 
**APPLICATION USAGE**

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.

The `pthread_rwlock_timedwrlock()` function is part of the Threads and Timeouts options and need not be provided on all implementations.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`pthread_rwlock_destroy()`, `pthread_rwlock_rdlock()`, `pthread_rwlock_timedrdlock()`,
`pthread_rwlock_tryrdlock()`, `pthread_rwlock_trywrlock()`, `pthread_rwlock_unlock()`,
`pthread_rwlock_wrlock()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`, `<time.h>`

**CHANGE HISTORY**

NAME
pthread_rwlock_tryrdlock — lock a read-write lock object for reading

SYNOPSIS
THR

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

DESCRIPTION
Refer to `pthread_rwlock_rdlock()`.
NAME
   pthread_rwlock_trywrlock, pthread_rwlock_wrlock — lock a read-write lock object for writing

SYNOPSIS
   #include <pthread.h>

   int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
   int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

DESCRIPTION
   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).

   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).

   Implementations may favor writers over readers to avoid writer starvation.

   Results are undefined if any of these functions are called with an uninitialized read-write lock.

   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.

RETURN VALUE
   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.

   If successful, the pthread_rwlock_wrlock() function shall return zero; otherwise, an error number
   shall be returned to indicate the error.

ERRORS
   The pthread_rwlock_trywrlock() function shall fail if:

   [EBUSY]   The read-write lock could not be acquired for writing because it was already
              locked for reading or writing.

   The pthread_rwlock_trywrlock() and pthread_rwlock_wrlock() functions may fail if:

   [EINVAL]  The value specified by rwlock does not refer to an initialized read-write lock
              object.

   The pthread_rwlock_wrlock() function may fail if:

   [EDEADLK] The current thread already owns the read-write lock for writing or reading.

   These functions shall not return an error code of [EINTR].
EXAMPLES
None.

APPLICATION USAGE
Applications using these functions may be subject to priority inversion, as discussed in the Base

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
pthread_rwlock_destroy(), pthread_rwlock_rdlock(), pthread_rwlock_timedrdlock(),
pthread_rwlock_timedwrlock(), pthread_rwlock_tryrdlock(), pthread_rwlock_trywrlock(), pthread_rwlock_unlock(), the Base
Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY
First released in Issue 5.

Issue 6
The following changes are made for alignment with IEEE Std 1003.1j-2000:

• 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).

• The [EDEADLK] error is deleted as a pthread_rwlock_trywrlock() error.

• The SEE ALSO section is updated.
NAME
pthread_rwlock_unlock — unlock a read-write lock object

SYNOPSIS
THR
#include <pthread.h>

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

DESCRIPTION
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.

If this function is called to release a read lock from the read-write lock object and there are other read locks currently held on this read-write lock object, the read-write lock object remains in the read locked state. If this function releases the last read lock for this read-write lock object, the read-write lock object shall be put in the unlocked state with no owners.

If this function is called to release a write lock for this read-write lock object, the read-write lock object shall be put in the unlocked state.

If there are threads blocked on the lock when it becomes available, the scheduling policy shall determine which thread(s) shall acquire the lock. If the Thread Execution Scheduling option is supported, when threads executing with the scheduling policies SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC are waiting on the lock, they shall acquire the lock in priority order when the lock becomes available. For equal priority threads, write locks shall take precedence over read locks. If the Thread Execution Scheduling option is not supported, it is implementation-defined whether write locks take precedence over read locks.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

RETURN VALUE
If successful, the pthread_rwlock_unlock() function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS
The pthread_rwlock_unlock() function may fail if:

[EINVAL] The value specified by rwlock does not refer to an initialized read-write lock object.

[EPERM] The current thread does not hold a lock on the read-write lock.

The pthread_rwlock_unlock() function shall not return an error code of [EINTR].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO

pthread_rwlock_destroy(), pthread_rwlock_rdlock(), pthread_rwlock_timedrdlock(),
pthread_rwlock_timedwrlock(), pthread_rwlock_tryrdlock(), pthread_rwlock_trywrlock(),
pthread_rwlock_wrlock(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

• The margin code in the SYNOPTIS 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).

• The DESCRIPTION is updated as follows:
  — The conditions under which writers have precedence over readers are specified.
  — The concept of read-write lock owner is deleted.

• The SEE ALSO section is updated.
**NAME**

pthread_rwlock_wrlock — lock a read-write lock object for writing

**SYNOPSIS**

THR

```c
#include <pthread.h>

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

**DESCRIPTION**

Refer to `pthread_rwlock_trywrlock()`. 
NAME

pthread_rwlockattr_destroy, pthread_rwlockattr_init — destroy and initialize the read-write lock attributes object

SYNOPSIS

THR

#include <pthread.h>

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

DESCRIPTION

The `pthread_rwlockattr_destroy()` function shall destroy a read-write lock attributes object. A destroyed `attr` attributes object can be reinitialized using `pthread_rwlockattr_init()`; the results of otherwise referencing the object after it has been destroyed are undefined. An implementation may cause `pthread_rwlockattr_destroy()` to set the object referenced by `attr` to an invalid value.

The `pthread_rwlockattr_init()` function shall initialize a read-write lock attributes object `attr` with the default value for all of the attributes defined by the implementation.

Results are undefined if `pthread_rwlockattr_init()` is called specifying an already initialized `attr` attributes object.

After a read-write lock attributes object has been used to initialize one or more read-write locks, any function affecting the attributes object (including destruction) shall not affect any previously initialized read-write locks.

RETURN VALUE

If successful, the `pthread_rwlockattr_destroy()` and `pthread_rwlockattr_init()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The `pthread_rwlockattr_destroy()` function may fail if:

[EINVAL] The value specified by `attr` is invalid.

The `pthread_rwlockattr_init()` function shall fail if:

[ENOMEM] Insufficient memory exists to initialize the read-write lock attributes object.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`pthread_rwlock_destroy()`, `pthread_rwlockattr_getpshared()`, `pthread_rwlockattr_setpshared()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`
CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- 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).

- The SEE ALSO section is updated.
# pthread_rwlockattr_getpshared()

## NAME

pthread_rwlockattr_getpshared, pthread_rwlockattr_setpshared — get and set the process-shared attribute of the read-write lock attributes object

## SYNOPSIS

```c
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t * restrict attr, int *restrict pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

## DESCRIPTION

The `pthread_rwlockattr_getpshared()` function shall obtain the value of the `process-shared` attribute from the initialized attributes object referenced by `attr`. The `pthread_rwlockattr_setpshared()` function shall set the `process-shared` attribute in an initialized attributes object referenced by `attr`.

The `process-shared` attribute shall be set to PTHREAD_PROCESS_SHARED to permit a read-write lock to be operated upon by any thread that has access to the memory where the read-write lock is allocated, even if the read-write lock is allocated in memory that is shared by multiple processes. If the `process-shared` attribute is PTHREAD_PROCESS_PRIVATE, the read-write lock shall only be operated upon by threads created within the same process as the thread that initialized the read-write lock; if threads of differing processes attempt to operate on such a read-write lock, the behavior is undefined. The default value of the `process-shared` attribute shall be PTHREAD_PROCESS_PRIVATE.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

## RETURN VALUE

Upon successful completion, the `pthread_rwlockattr_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_rwlockattr_setpshared()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

## ERRORS

The `pthread_rwlockattr_getpshared()` and `pthread_rwlockattr_setpshared()` functions may fail if:

- [EINVAL] The value specified by `attr` is invalid.
- The `pthread_rwlockattr_setpshared()` function may fail if:
  - [EINVAL] The new value specified for the attribute is outside the range of legal values for that attribute.

These functions shall not return an error code of [EINTR].
EXCEPTIONS

APPLICATION USAGE

RATIONALE

FUTURE DIRECTIONS

SEE ALSO

CHANGE HISTORY

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

• 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).

• The DESCRIPTION notes that additional attributes are implementation-defined.

• The SEE ALSO section is updated.

The restrict keyword is added to the `pthread_rwlockattr_getpshared()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
pthread_rwlockattr_init — initialize the read-write lock attributes object

SYNOPSIS
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

DESCRIPTION
Refer to pthread_rwlockattr_destroy().
NAME

pthread_rwlockattr_setpshared — set the process-shared attribute of the read-write lock attributes object

SYNOPSIS

```c
#include <pthread.h>

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

DESCRIPTION

Refer to `pthread_rwlockattr_getpshared()`.
SYSTEM INTERFACES

NAME

pthread_self — get the calling thread ID

SYNOPSIS

THR

#include <pthread.h>

pthread_t pthread_self(void);

DESCRIPTION

The pthread_self() function shall return the thread ID of the calling thread.

RETURN VALUE

Refer to the DESCRIPTION.

ERRORS

No errors are defined.

The pthread_self() function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The pthread_self() function provides a capability similar to the getpid() function for processes and the rationale is the same: the creation call does not provide the thread ID to the created thread.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_create(), pthread_equal(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The pthread_self() function is marked as part of the Threads option.
NAME
pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel — set cancelability state

SYNOPSIS
THR
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);

DESCRIPTION
The pthread_setcancelstate() function shall atomically both set the calling thread’s cancelability state to the indicated state and return the previous cancelability state at the location referenced by oldstate. Legal values for state are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DISABLE.
The pthread_setcanceltype() function shall atomically both set the calling thread’s cancelability type to the indicated type and return the previous cancelability type at the location referenced by oldtype. Legal values for type are PTHREAD_CANCEL_DEFERRED and PTHREAD_CANCEL_ASYNCHRONOUS.
The cancelability state and type of any newly created threads, including the thread in which main() was first invoked, shall be PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DEFERRED respectively.
The pthread_testcancel() function shall create a cancelation point in the calling thread. The pthread_testcancel() function shall have no effect if cancelability is disabled.

RETURN VALUE
If successful, the pthread_setcancelstate() and pthread_setcanceltype() functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS
The pthread_setcancelstate() function may fail if:
[EINVAL] The specified state is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.
The pthread_setcanceltype() function may fail if:
[EINVAL] The specified type is not PTHREAD_CANCEL_DEFERRED or PTHREAD_CANCEL_ASYNCHRONOUS.
These functions shall not return an error code of [EINTR].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
The pthread_setcancelstate() and pthread_setcanceltype() functions control the points at which a thread may be asynchronously canceled. For cancelation control to be usable in modular fashion, some rules need to be followed.
An object can be considered to be a generalization of a procedure. It is a set of procedures and global variables written as a unit and called by clients not known by the object. Objects may depend on other objects.
First, cancelability should only be disabled on entry to an object, never explicitly enabled. On exit from an object, the cancelability state should always be restored to its value on entry to the object.

This follows from a modularity argument: if the client of an object (or the client of an object that uses that object) has disabled cancelability, it is because the client does not want to be concerned about cleaning up if the thread is canceled while executing some sequence of actions. If an object is called in such a state and it enables cancelability and a cancelation request is pending for that thread, then the thread is canceled, contrary to the wish of the client that disabled.

Second, the cancelability type may be explicitly set to either deferred or asynchronous upon entry to an object. But as with the cancelability state, on exit from an object the cancelability type should always be restored to its value on entry to the object.

Finally, only functions that are cancel-safe may be called from a thread that is asynchronously cancelable.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

pthread_cancel(), the Base Definitions volume of IEEE Std 1003.1-2001, <pthread.h>

**CHANGE HISTORY**

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

**Issue 6**

The pthread_setcancelstate(), pthread_setcanceltype(), and pthread_testcancel() functions are marked as part of the Threads option.
NAME

pthread_setconcurrency — set the level of concurrency

SYNOPSIS

XSI

```c
#include <pthread.h>

int pthread_setconcurrency(int new_level);
```

DESCRIPTION

Refer to `pthread_getconcurrency()`.
NAME

pthread_setschedparam — dynamic thread scheduling parameters access (REALTIME THREADS)

SYNOPSIS

#include <pthread.h>

int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);

DESCRIPTION

Refer to pthread_setschedparam().
NAME

pthread_setschedprio — dynamic thread scheduling parameters access (REALTIME THREADS)

SYNOPSIS

#include <pthread.h>

int pthread_setschedprio(pthread_t thread, int prio);

DESCRIPTION

The pthread_setschedprio() function shall set the scheduling priority for the thread whose thread ID is given by thread to the value given by prio. See Scheduling Policies (on page 44) for a description on how this function call affects the ordering of the thread in the thread list for its new priority.

If the pthread_setschedprio() function fails, the scheduling priority of the target thread shall not be changed.

RETURN VALUE

If successful, the pthread_setschedprio() function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The pthread_setschedprio() function may fail if:

[EINVAL] The value of prio is invalid for the scheduling policy of the specified thread.

[ENOTSUP] An attempt was made to set the priority to an unsupported value.

[EPERM] The caller does not have the appropriate permission to set the scheduling policy of the specified thread.

[EPERM] The implementation does not allow the application to modify the priority to the value specified.

[ESRCH] The value specified by thread does not refer to an existing thread.

The pthread_setschedprio() function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The pthread_setschedprio() function provides a way for an application to temporarily raise its priority and then lower it again, without having the undesired side effect of yielding to other threads of the same priority. This is necessary if the application is to implement its own strategies for bounding priority inversion, such as priority inheritance or priority ceilings. This capability is especially important if the implementation does not support the Thread Priority Protection or Thread Priority Inheritance options, but even if those options are supported it is needed if the application is to bound priority inheritance for other resources, such as semaphores.

The standard developers considered that while it might be preferable conceptually to solve this problem by modifying the specification of pthread_setschedparam(), it was too late to make such a change, as there may be implementations that would need to be changed. Therefore, this new function was introduced.
FUTURE DIRECTIONS

None.

SEE ALSO
Scheduling Policies (on page 44), pthread_setschedparam(), the Base Definitions volume of
IEEE Std 1003.1-2001, <pthread.h>

CHANGE HISTORY
First released in Issue 6. Included as a response to IEEE PASC Interpretation 1003.1 #96.
NAME
pthread_setspecific — thread-specific data management

SYNOPSIS
THR
#include <pthread.h>

int pthread_setspecific(pthread_key_t key, const void *value);

DESCRIPTION
Refer to pthread_getspecific().
NAME

pthread_sigmask, sigprocmask — examine and change blocked signals

SYNOPSIS

```c
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set, 
                    sigset_t *restrict oset);

int sigprocmask(int how, const sigset_t *restrict set, 
                 sigset_t *restrict oset);
```

DESCRIPTION

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.

In a single-threaded process, the *sigprocmask()* function shall examine or change (or both) the signal mask of the calling thread.

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.

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:

- SIG_BLOCK The resulting set shall be the union of the current set and the signal set pointed to by *set*.
- SIG_SETMASK The resulting set shall be the signal set pointed to by *set*.
- SIG_UNBLOCK The resulting set shall be the intersection of the current set and the complement of the signal set pointed to by *set*.

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 process’ signal mask shall be unchanged; thus the call can be used to enquire about currently blocked signals.

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.

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.

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.

If *sigprocmask()* fails, the thread’s signal mask shall not be changed.

The use of the *sigprocmask()* function is unspecified in a multi-threaded process.

RETURN VALUE

Upon successful completion *pthread_sigmask()* shall return 0; otherwise, it shall return the corresponding error number.

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.
The `pthread_sigmask()` function shall fail if:

- `[EINVAL]` The value of the `how` argument is not equal to one of the defined values.

The `pthread_sigmask()` function shall not return an error code of `[EINTR]`.

None.

None.

When a process’ signal mask is changed in a signal-catching function that is installed by `sigaction()`, the restoration of the signal mask on return from the signal-catching function overrides that change (see `sigaction()`). If the signal-catching function was installed with `signal()`, it is unspecified whether this occurs.

See `kill()` for a discussion of the requirement on delivery of signals.

None.

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

The `pthread_sigmask()` function is added for alignment with the POSIX Threads Extension.

The `pthread_sigmask()` function is marked as part of the Threads option.

The SYNSOPSIS for `sigprocmask()` is marked as a CX extension to note that the presence of this function in the `<signal.h>` header is an extension to the ISO C standard.

The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- The DESCRIPTION is updated to explicitly state the functions which may generate the signal.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The `restrict` keyword is added to the `pthread_sigmask()` and `sigprocmask()` prototypes for alignment with the ISO/IEC 9899:1999 standard.
**NAME**

pthread_spin_destroy, pthread_spin_init — destroy or initialize a spin lock object (ADVANCED REALTIME THREADS)

**SYNOPSIS**

```c
#include <pthread.h>

int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

**DESCRIPTION**

The `pthread_spin_destroy()` function shall destroy the spin lock referenced by `lock` 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_spin_init()`. The results are undefined if `pthread_spin_destroy()` is called when a thread holds the lock, or if this function is called with an uninitialized thread spin lock.

The `pthread_spin_init()` function shall allocate any resources required to use the spin lock referenced by `lock` and initialize the lock to an unlocked state.

TSH If the Thread Process-Shared Synchronization option is supported and the value of `pshared` is PTHREAD_PROCESS_SHARED, the implementation shall permit the spin lock to be operated upon by any thread that has access to the memory where the spin lock is allocated, even if it is allocated in memory that is shared by multiple processes.

If the Thread Process-Shared Synchronization option is supported and the value of `pshared` is PTHREAD_PROCESS_PRIVATE, or if the option is not supported, the spin lock shall only be operated upon by threads created within the same process as the thread that initialized the spin lock. If threads of differing processes attempt to operate on such a spin lock, the behavior is undefined.

The results are undefined if `pthread_spin_init()` is called specifying an already initialized spin lock. The results are undefined if a spin lock is used without first being initialized.

If the `pthread_spin_init()` function fails, the lock is not initialized and the contents of `lock` are undefined.

Only the object referenced by `lock` may be used for performing synchronization.

The result of referring to copies of that object in calls to `pthread_spin_destroy()`, `pthread_spin_lock()`, `pthread_spin_trylock()`, or `pthread_spin_unlock()` is undefined.

**RETURN VALUE**

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

**ERRORS**

These functions may fail if:

- **[EBUSY]** The implementation has detected an attempt to initialize or destroy a spin lock while it is in use (for example, while being used in a `pthread_spin_lock()` call) by another thread.

- **[EINVAL]** The value specified by `lock` is invalid.

The `pthread_spin_init()` function shall fail if:

- **[EAGAIN]** The system lacks the necessary resources to initialize another spin lock.
[ENOMEM] Insufficient memory exists to initialize the lock.

These functions shall not return an error code of [EINTR].

**EXAMPLES**
None.

**APPLICATION USAGE**
The `pthread_spin_destroy()` and `pthread_spin_init()` functions are part of the Spin Locks option and need not be provided on all implementations.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`pthread_spin_lock()`, `pthread_spin_unlock()`, the Base Definitions volume of IEEE Std 1003.1-2001,

**CHANGE HISTORY**

In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.
**NAME**

pthread_spin_lock, pthread_spin_trylock — lock a spin lock object (ADVANCED REALTIME THREADS)

**SYNOPSIS**

```c
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
```

**DESCRIPTION**

The `pthread_spin_lock()` function shall lock the spin lock referenced by `lock`. The calling thread shall acquire the lock if it is not held by another thread. Otherwise, the thread shall spin (that is, shall not return from the `pthread_spin_lock()` call) until the lock becomes available. The results are undefined if the calling thread holds the lock at the time the call is made. The `pthread_spin_trylock()` function shall lock the spin lock referenced by `lock` if it is not held by any thread. Otherwise, the function shall fail.

The results are undefined if any of these functions is called with an uninitialized spin lock.

**RETURN VALUE**

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

**ERRORS**

These functions may fail if:

- **EINVAL** The value specified by `lock` does not refer to an initialized spin lock object.
- **EDEADLK** The calling thread already holds the lock.
- **EBUSY** A thread currently holds the lock.

These functions shall not return an error code of **EINTR**.

**EXAMPLES**

None.

**APPLICATION USAGE**

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.

The `pthread_spin_lock()` and `pthread_spin_trylock()` functions are part of the Spin Locks option and need not be provided on all implementations.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`pthread_spin_destroy()`, `pthread_spin_unlock()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<pthread.h>`
**CHANGE HISTORY**


In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.
NAME

pthread_spin_unlock — unlock a spin lock object (ADVANCED REALTIME THREADS)

SYNOPSIS

#include <pthread.h>

int pthread_spin_unlock(pthread_spinlock_t *lock);

DESCRIPTION

The pthread_spin_unlock() function shall release the spin lock referenced by lock which was
locked via the pthread_spin_lock() or pthread_spin_trylock() functions. The results are undefined if
the lock is not held by the calling thread. If there are threads spinning on the lock when
pthread_spin_unlock() is called, the lock becomes available and an unspecified spinning thread
shall acquire the lock.

The results are undefined if this function is called with an uninitialized thread spin lock.

RETURN VALUE

Upon successful completion, the pthread_spin_unlock() function shall return zero; otherwise, an
error number shall be returned to indicate the error.

ERRORS

The pthread_spin_unlock() function may fail if:

EINVAL An invalid argument was specified.
EPERM The calling thread does not hold the lock.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The pthread_spin_unlock() function is part of the Spin Locks option and need not be provided on
all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_spin_destroy(), pthread_spin_lock(), the Base Definitions volume of IEEE Std 1003.1-2001,
pthread.h

CHANGE HISTORY

In the SYNOPSIS, the inclusion of <sys/types.h> is no longer required.
NAME
pthread_testcancel — set cancelability state

SYNOPSIS

THR
#include <pthread.h>

void pthread_testcancel(void);

DESCRIPTION
Refer to pthread_setcancelstate().
NAME
ptsname — get name of the slave pseudo-terminal device

SYNOPSIS
#include <stdlib.h>
char *ptsname(int fildes);

DESCRIPTION
The ptsname() function shall return the name of the slave pseudo-terminal device associated
with a master pseudo-terminal device. The fildes argument is a file descriptor that refers to the
master device. The ptsname() function shall return a pointer to a string containing the pathname
of the corresponding slave device.

The ptsname() function need not be reentrant. A function that is not required to be reentrant is
not required to be thread-safe.

RETURN VALUE
Upon successful completion, ptsname() shall return a pointer to a string which is the name of the
pseudo-terminal slave device. Upon failure, ptsname() shall return a null pointer. This could
occur if fildes is an invalid file descriptor or if the slave device name does not exist in the file
system.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
The value returned may point to a static data area that is overwritten by each call to ptsname().

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
grantpt(), open(), ttyname(), unlockpt(), the Base Definitions volume of IEEE Std 1003.1-2001,
<stdlib.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
A note indicating that this function need not be reentrant is added to the DESCRIPTION.
NAME
putc — put a byte on a stream

SYNOPSIS
#include <stdio.h>
int putc(int c, FILE *stream);

DESCRIPTION
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

The putc() function shall be equivalent to fputc(), except that if it is implemented as a macro it
may evaluate stream more than once, so the argument should never be an expression with side
effects.

RETURN VALUE
Refer to fputc().

ERRORS
Refer to fputc().

EXAMPLES
None.

APPLICATION USAGE
Since it may be implemented as a macro, putc() may treat a stream argument with side effects
incorrectly. In particular, putc(c,*f++) does not necessarily work correctly. Therefore, use of this
function is not recommended in such situations; fputc() should be used instead.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fputc(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
putc_unlocked — stdio with explicit client locking

SYNOPSIS
#include <stdio.h>

int putc_unlocked(int c, FILE *stream);

DESCRIPTION
Refer to getc_unlocked().
NAME
putchar — put a byte on a stdout stream

SYNOPSIS
#include <stdio.h>
int putchar(int c);

DESCRIPTION
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 function call putchar(c) shall be equivalent to putc(c, stdout).

RETURN VALUE
Refer to fputc().

ERRORS
Refer to fputc().

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
putc(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
putchar_unlocked — stdio with explicit client locking

SYNOPSIS
#include <stdio.h>

int putchar_unlocked(int c);

DESCRIPTION
Refer to getc_unlocked().
NAME
putenv — change or add a value to an environment

SYNOPSIS
XSI
#include <stdlib.h>

int putenv(char *string);

DESCRIPTION
The putenv() function shall use the string argument to set environment variable values. The string argument should point to a string of the form "name=value. The putenv() function shall make the value of the environment variable name equal to value by altering an existing variable or creating a new one. In either case, the string pointed to by string shall become part of the environment, so altering the string shall change the environment. The space used by string is no longer used once a new string-defining name is passed to putenv().

The putenv() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE
Upon successful completion, putenv() shall return 0; otherwise, it shall return a non-zero value and set errno to indicate the error.

ERRORS
The putenv() function may fail if:

[ENOMEM] Insufficient memory was available.

EXAMPLES
Changing the Value of an Environment Variable
The following example changes the value of the HOME environment variable to the value /usr/home.

#include <stdlib.h>
...
static char *var = "HOME=/usr/home";
int ret;
ret = putenv(var);

APPLICATION USAGE
The putenv() function manipulates the environment pointed to by environ, and can be used in conjunction with getenv().

This routine may use malloc() to enlarge the environment.

A potential error is to call putenv() with an automatic variable as the argument, then return from the calling function while string is still part of the environment.

The setenv() function is preferred over this function.

RATIONALE
The standard developers noted that putenv() is the only function available to add to the environment without permitting memory leaks.
FUTURE DIRECTIONS
None.

SEE ALSO
exec, getenv(), malloc(), setenv(), the Base Definitions volume of IEEE Std 1003.1-2001, stdlib.h

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The type of the argument to this function is changed from const char * to char *. This was indicated as a FUTURE DIRECTION in previous issues.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.
putmsg()

NAME
putmsg, putpmsg — send a message on a STREAM (STREAMS)

SYNOPSIS
XSR

```c
#include <stropts.h>

int putmsg(int fildes, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flags);
int putpmsg(int fildes, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int band, int flags);
```

DESCRIPTION
The `putmsg()` function shall create a message from a process buffer(s) and send the message to a STREAMS file. The message may contain either a data part, a control part, or both. The data and control parts are distinguished by placement in separate buffers, as described below. The semantics of each part are defined by the STREAMS module that receives the message.

The `putpmsg()` function is equivalent to `putmsg()`, except that the process can send messages in different priority bands. Except where noted, all requirements on `putmsg()` also pertain to `putpmsg()`.

The `fildes` argument specifies a file descriptor referencing an open STREAM. The `ctlptr` and `dataptr` arguments each point to a `strbuf` structure.

The `ctlptr` argument points to the structure describing the control part, if any, to be included in the message. The `buf` member in the `strbuf` structure points to the buffer where the control information resides, and the `len` member indicates the number of bytes to be sent. The `maxlen` member is not used by `putmsg()`. In a similar manner, the argument `dataptr` specifies the data, if any, to be included in the message. The `flags` argument indicates what type of message should be sent and is described further below.

To send the data part of a message, the application shall ensure that `dataptr` is not a null pointer and the `len` member of `dataptr` is 0 or greater. To send the control part of a message, the application shall ensure that the corresponding values are set for `ctlptr`. No data (control) part shall be sent if either `dataptr` (`ctlptr`) is a null pointer or the `len` member of `dataptr` (`ctlptr`) is set to −1.

For `putmsg()`, if a control part is specified and `flags` is set to RS_HIPRI, a high priority message shall be sent. If no control part is specified, and `flags` is set to RS_HIPRI, `putmsg()` shall fail and set `errno` to [EINVAL]. If `flags` is set to 0, a normal message (priority band equal to 0) shall be sent. If a control part and data part are not specified and `flags` is set to 0, no message shall be sent and 0 shall be returned.

For `putpmsg()`, the flags are different. The `flags` argument is a bitmask with the following mutually-exclusive flags defined: MSG_HIPRI and MSG_BAND. If `flags` is set to 0, `putpmsg()` shall fail and set `errno` to [EINVAL]. If a control part is specified and `flags` is set to MSG_HIPRI and `band` is set to 0, a high-priority message shall be sent. If `flags` is set to MSG_HIPRI and either no control part is specified or `band` is set to a non-zero value, `putpmsg()` shall fail and set `errno` to [EINVAL]. If `flags` is set to MSG_BAND, then a message shall be sent in the priority band specified by `band`. If a control part and data part are not specified and `flags` is set to MSG_BAND, no message shall be sent and 0 shall be returned.

The `putmsg()` function shall block if the STREAM write queue is full due to internal flow control conditions, with the following exceptions:

- For high-priority messages, `putmsg()` shall not block on this condition and continues processing the message.
• For other messages, putmsg() shall not block but shall fail when the write queue is full and O_NONBLOCK is set.

The putmsg() function shall also block, unless prevented by lack of internal resources, while waiting for the availability of message blocks in the STREAM, regardless of priority or whether O_NONBLOCK has been specified. No partial message shall be sent.

RETURN VALUE
Upon successful completion, putmsg() and putpmsg() shall return 0; otherwise, they shall return –1 and set errno to indicate the error.

ERRORS
The putmsg() and putpmsg() functions shall fail if:

[EAGAIN] A non-priority message was specified, the O_NONBLOCK flag is set, and the STREAM write queue is full due to internal flow control conditions; or buffers could not be allocated for the message that was to be created.

[EBADF] fildes is not a valid file descriptor open for writing.

[EINTR] A signal was caught during putmsg().

[EINVAL] An undefined value is specified in flags, or flags is set to RS_HIPRI or MSG_HIPRI and no control part is supplied, or the STREAM or multiplexer referenced by fildes is linked (directly or indirectly) downstream from a multiplexer, or flags is set to MSG_HIPRI and band is non-zero (for putpmsg() only).

[ENOSR] Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.

[ENOSTR] A STREAM is not associated with fildes.

[ENXIO] A hangup condition was generated downstream for the specified STREAM.

[EPIPE] or [EIO] The fildes argument refers to a STREAMS-based pipe and the other end of the pipe is closed. A SIGPIPE signal is generated for the calling thread.

[ERANGE] The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

In addition, putmsg() and putpmsg() shall fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of putmsg() or putpmsg(), but reflects the prior error.
EXAMPLES

Sending a High-Priority Message

The value of $fd$ is assumed to refer to an open STREAMS file. This call to \texttt{putmsg()} does the following:

1. Creates a high-priority message with a control part and a data part, using the buffers pointed to by $ctrlbuf$ and $databuf$, respectively.

2. Sends the message to the STREAMS file identified by $fd$.

\begin{verbatim}
#include <stropts.h>
#include <string.h>
...
int fd;
char *ctrlbuf = "This is the control part";
char *databuf = "This is the data part";
struct strbuf ctrl;
struct strbuf data;
int ret;
ctrl.buf = ctrlbuf;
ctrl.len = strlen(ctrlbuf);
data.buf = databuf;
data.len = strlen(databuf);
ret = putmsg(fd, &ctrl, &data, MSG_HIPRI);
\end{verbatim}

Using putpmsg()

This example has the same effect as the previous example. In this example, however, the \texttt{putpmsg()} function creates and sends the message to the STREAMS file.

\begin{verbatim}
#include <stropts.h>
#include <string.h>
...
int fd;
char *ctrlbuf = "This is the control part";
char *databuf = "This is the data part";
struct strbuf ctrl;
struct strbuf data;
int ret;
ctrl.buf = ctrlbuf;
ctrl.len = strlen(ctrlbuf);
data.buf = databuf;
data.len = strlen(databuf);
ret = putpmsg(fd, &ctrl, &data, 0, MSG_HIPRI);
\end{verbatim}

APPLICATION USAGE

None.
putmsg()

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.6 (on page 38), getmsg(), poll(), read(), write(), the Base Definitions volume of IEEE Std 1003.1-2001, <stropts.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

The following text is removed from the DESCRIPTION: “The STREAM head guarantees that the control part of a message generated by putmsg() is at least 64 bytes in length.”

Issue 6
This function is marked as part of the XSI STREAMS Option Group.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME

puts — put a string on standard output

SYNOPSIS

#include <stdio.h>

int puts(const char *s);

DESCRIPTION

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 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.

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().

RETURN VALUE

Upon successful completion, puts() shall return a non-negative number. Otherwise, it shall return EOF, shall set an error indicator for the stream, and errno shall be set to indicate the error.

ERRORS

Refer to fputc().

EXAMPLES

Printing to Standard Output

The following example gets the current time, converts it to a string using localtime() and asctime(), and prints it to standard output using puts(). It then prints the number of minutes to an event for which it is waiting.

#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n", minutes_to_event);
...

APPLICATION USAGE

The puts() function appends a <newline>, while fputs() does not.

RATIONALE

None.

FUTURE DIRECTIONS

None.
SEE ALSO

fopen(), fputs(), putc(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Extensions beyond the ISO C standard are marked.
pututxline() — put an entry into the user accounting database

SYNOPSIS

```c
#include <utmpx.h>

struct utmpx *pututxline(const struct utmpx *utmpx);
```

DESCRIPTION

Refer to `endutxent()`.
NAME
putwc — put a wide character on a stream

SYNOPSIS
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wchar_t wc, FILE *stream);

DESCRIPTION
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

The putwc() function shall be equivalent to fputwc(), except that if it is implemented as a macro
it may evaluate stream more than once, so the argument should never be an expression with side
effects.

RETURN VALUE
Refer to fputwc().

ERRORS
Refer to fputwc().

APPLICATION USAGE
Since it may be implemented as a macro, putwc() may treat a stream argument with side effects
incorrectly. In particular, putwc(wc,"f++") need not work correctly. Therefore, use of this function
is not recommended; fputwc() should be used instead.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fputwc(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>, <wchar.h>

CHANGE HISTORY
First released as a World-wide Portability Interface in Issue 4.

Issue 5
Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument wc
is changed from wint_t to wchar_t.

The Optional Header (OH) marking is removed from <stdio.h>.
putwchar

NAME
putwchar — put a wide character on a stdout stream

SYNOPSIS
#include <wchar.h>

wint_t putwchar(wchar_t wc);

DESCRIPTION
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

The function call putwchar(wc) shall be equivalent to putwc(wc,stdout).

RETURN VALUE
Refer to fputwc().

ERRORS
Refer to fputwc().

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fputwc(), putwc(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4.

Issue 5
Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument wc
is changed from wint_t to wchar_t.
NAME
pwrite — write on a file

SYNOPSIS
#include <unistd.h>

ssize_t pwrite(int fildes, const void *buf, size_t nbyte, off_t offset);

DESCRIPTION
Refer to write().
NAME
qsort — sort a table of data

SYNOPSIS
#include <stdlib.h>
void qsort(void *base, size_t nel, size_t width,
int (*compar)(const void *, const void *));

DESCRIPTION
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
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.
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.

RETURN VALUE
The qsort() function shall not return a value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
The comparison function need not compare every byte, so arbitrary data may be contained in
the elements in addition to the values being compared.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
raise — send a signal to the executing process

SYNOPSIS
#include <signal.h>
int raise(int sig);

DESCRIPTION
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 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.

If the implementation supports the Threads option, the effect of the raise() function shall be equivalent to calling:

pthread_kill(pthread_self(), sig);

Otherwise, the effect of the raise() function shall be equivalent to calling:

kill(getpid(), sig);

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, a non-zero value shall be returned and errno shall be set to indicate the error.

ERRORS
The raise() function shall fail if:

EINVAL The value of the sig argument is an invalid signal number.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
The term “thread” is an extension to the ISO C standard.

FUTURE DIRECTIONS
None.

SEE ALSO
kill(), sigaction(), the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>, <sys/types.h>

CHANGE HISTORY
First released in Issue 4. Derived from the ANSI C standard.

Issue 5
The DESCRIPTION is updated for alignment with the POSIX Threads Extension.
Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the requirement to set `errno` on error is added.
- The `[EINVAL]` error condition is added.
NAME
rand, rand_r, srand — pseudo-random number generator

SYNOPSIS
#include <stdlib.h>
int rand(void);

TSF
int rand_r(unsigned *seed);
void srand(unsigned seed);

DESCRIPTION
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 rand() function shall compute a sequence of pseudo-random integers in the range \([0, \text{RAND_MAX}]\) with a period of at least \(2^{32}\).
The rand() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.
The rand_r() function shall compute a sequence of pseudo-random integers in the range \([0, \text{RAND_MAX}]\). (The value of the \text{RAND_MAX} macro shall be at least 32,767.) 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.
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.
The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-2001 calls rand() or srand().

RETURN VALUE
The rand() function shall return the next pseudo-random number in the sequence.
The rand_r() function shall return a pseudo-random integer.
The srand() function shall not return a value.

ERRORS
No errors are defined.

EXAMPLES
Generating a Pseudo-Random Number Sequence
The following example demonstrates how to generate a sequence of pseudo-random numbers.

```
#include <stdio.h>
#include <stdlib.h>
...
long count, i;
char *keystr;
int elementlen, len;
char c;
```
... Initial random number generator. */
    srand(1);
    /* Create keys using only lowercase characters */
    len = 0;
    for (i=0; i<count; i++) {
        while (len < elementlen) {
            c = (char) (rand() % 128);
            if (islower(c))
                keystr[len++] = c;
        }
        keystr[len] = '\0';
        printf("%s Element%0*ld\n", keystr, elementlen, i);
        len = 0;
    }

Generating the Same Sequence on Different Machines

The following code defines a pair of functions that could be incorporated into applications
wishing to ensure that the same sequence of numbers is generated across different machines.

static unsigned long next = 1;
int myrand(void) /* RAND_MAX assumed to be 32767. */
{
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}
void mysrand(unsigned seed)
{
    next = seed;
}

APPLICATION USAGE

The \texttt{drand48()} function provides a much more elaborate random number generator.

The limitations on the amount of state that can be carried between one function call and another
mean the \texttt{rand_r()} function can never be implemented in a way which satisfies all of the
requirements on a pseudo-random number generator. Therefore this function should be avoided
whenever non-trivial requirements (including safety) have to be fulfilled.

RATIONALE

The ISO C standard \texttt{rand()} and \texttt{srand()} functions allow per-process pseudo-random streams
shared by all threads. Those two functions need not change, but there has to be mutual-
exclusion that prevents interference between two threads concurrently accessing the random
number generator.

With regard to \texttt{rand()}, there are two different behaviors that may be wanted in a multi-threaded
program:

1. A single per-process sequence of pseudo-random numbers that is shared by all threads
   that call \texttt{rand()}
2. A different sequence of pseudo-random numbers for each thread that calls \texttt{rand()}
This is provided by the modified thread-safe function based on whether the seed value is global
to the entire process or local to each thread.

This does not address the known deficiencies of the `rand()` function implementations, which
have been approached by maintaining more state. In effect, this specifies new thread-safe forms
of a deficient function.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`drand48()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**
The `rand_r()` function is included for alignment with the POSIX Threads Extension.
A note indicating that the `rand()` function need not be reentrant is added to the DESCRIPTION.

**Issue 6**
Extensions beyond the ISO C standard are marked.
The `rand_r()` function is marked as part of the Thread-Safe Functions option.
random()

NAME
random — generate pseudo-random number

SYNOPSIS
#include <stdlib.h>
long random(void);

DESCRIPTION
Refer to initstate().
NAME
pread, read — read from a file

SYNOPSIS
#include <unistd.h>

XSI
ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
ssize_t read(int fildes, void *buf, size_t nbyte);

DESCRIPTION
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.

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.

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.

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.

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.

If the value of nbyte is greater than {SSIZE_MAX}, the result is implementation-defined.

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.

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.

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, lseek()
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.

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
read()

file is less than nbyte, if the read() request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than nbyte bytes immediately available for reading. For example, a read() from a file associated with a terminal may return one typed line of data.

If a read() is interrupted by a signal before it reads any data, it shall return -1 with errno set to [EINVAL].

If a read() is interrupted by a signal after it has successfully read some data, it shall return the number of bytes read.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with fildes.

If fildes refers to a socket, read() shall be equivalent to recv() with no flags set.

If the O_DSYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion. If the O_SYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

If fildes refers to a shared memory object, the result of the read() function is unspecified.

If fildes refers to a typed memory object, the result of the read() function is unspecified.

A read() from a STREAMS file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default shall be byte-stream mode. This can be changed using the _SRDOPT ioctl() request, and can be tested with _GRDOPT ioctl().

In byte-stream mode, read() shall retrieve data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, read() shall retrieve data until as many bytes as were requested are transferred, or until a message boundary is reached. If read() does not retrieve all the data in a message, the remaining data shall be left on the STREAM, and can be retrieved by the next read() call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the read() returns shall be discarded, and shall not be available for a subsequent read(), getmsg(), or getpmsg() call.

How read() handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, read() shall accept data until it has read nbyte bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The read() function shall then return the number of bytes read, and place the zero-byte message back on the STREAM to be retrieved by the next read(), getmsg(), or getpmsg(). In message-nondiscard mode or message-discard mode, a zero-byte message shall return 0 and the message shall be removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message shall be removed from the STREAM and 0 shall be returned, regardless of the read mode.

A read() from a STREAMS file shall return the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMs are in control-normal mode, in which a read() from a STREAMS file can only process messages that contain a data part but do not contain a control part. The read() shall fail if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the _SRDOPT ioctl() command. In control-data mode, read() shall convert any control part to data and pass it to the application before passing any data part originally present...
in the same message. In control-discard mode, read() shall discard message control parts but
return to the process any data part in the message.

In addition, read() shall fail if the STREAM head had processed an asynchronous error before the
3621 call. In this case, the value of errno shall not reflect the result of read(), but reflect the prior error.
If a hangup occurs on the STREAM being read, read() shall continue to operate normally until
3624 the STREAM head read queue is empty. Thereafter, it shall return 0.

The pread() function shall be equivalent to read(), except that it shall read from a given position
3627 in the file without changing the file pointer. The first three arguments to pread() are the same as
read() with the addition of a fourth argument offset for the desired position inside the file. An
3628 attempt to perform a pread() on a file that is incapble of seeking shall result in an error.

RETURN VALUE
3630 Upon successful completion, read() and pread() shall return a non-negative integer indicating the
or number of bytes actually read. Otherwise, the functions shall return −1 and set errno to indicate
the error.

ERRORS
3633 The read() and pread() functions shall fail if:

3635 [EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the process would be
delayed.
3636
3637 [EBADF] The fildes argument is not a valid file descriptor open for reading.
3638
3639 [EBADMSG] The file is a STREAM file that is set to control-normal mode and the message
waiting to be read includes a control part.
3640
3641 [EINVAL] The STREAM or multiplexer referenced by fildes is linked (directly or
indirectly) downstream from a multiplexer.
3642
3643 [EIO] The process is a member of a background process attempting to read from its
controlling terminal, the process is ignoring or blocking the SIGTTIN signal,
or the process group is orphaned. This error may also be generated for
implementation-defined reasons.
3644
3645 [EINVAL] The fildes argument refers to a directory and the implementation does not
allow the directory to be read using read() or pread(). The readdir() function
should be used instead.
3646
3647 [EOVERFLOW] The file is a regular file, nbyte is greater than 0, the starting position is before
the end-of-file, and the starting position is greater than or equal to the offset
maximum established in the open file description associated with fildes.
3648
The read() function shall fail if:

3650 [EAGAIN] or [EWOULDBLOCK]
3651 The file descriptor is for a socket, is marked O_NONBLOCK, and no data is
waiting to be received.
3652
3653 [ECONNRESET] A read was attempted on a socket and the connection was forcibly closed by
its peer.
3654
3655 [ENOTCONN] A read was attempted on a socket that is not connected.
3656
3657 [ETIMEDOUT] A read was attempted on a socket and a transmission timeout occurred.
The `read()` and `pread()` functions may fail if:

- `[EIO]` A physical I/O error has occurred.
- `[ENOBUFFS]` Insufficient resources were available in the system to perform the operation.
- `[ENOMEM]` Insufficient memory was available to fulfill the request.
- `[ENXIO]` A request was made of a nonexistent device, or the request was outside the capabilities of the device.

The `pread()` function shall fail, and the file pointer shall remain unchanged, if:

- `[EINVAL]` The `offset` argument is invalid. The value is negative.
- `[EOVERFLOW]` The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the file.
- `[ENXIO]` A request was outside the capabilities of the device.
- `[ESPIPE]` `fildes` is associated with a pipe or FIFO.

**EXAMPLES**

**Reading Data into a Buffer**

The following example reads data from the file associated with the file descriptor `fd` into the buffer pointed to by `buf`.

```c
#include <sys/types.h>
#include <unistd.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_read;
int fd;
...
nbytes = sizeof(buf);
bytes_read = read(fd, buf, nbytes);
...```

**APPLICATION USAGE**

None.

**RATIONALE**

This volume of IEEE Std 1003.1-2001 does not specify the value of the file offset after an error is returned; there are too many cases. For programming errors, such as `[EBADF]`, the concept is meaningless since no file is involved. For errors that are detected immediately, such as `[EAGAIN]`, clearly the pointer should not change. After an interrupt or hardware error, however, an updated value would be very useful and is the behavior of many implementations.

Note that a `read()` of zero bytes does not modify `st_atime`. A `read()` that requests more than zero bytes, but returns zero, shall modify `st_atime`.

Implementations are allowed, but not required, to perform error checking for `read()` requests of zero bytes.
Input and Output

The use of I/O with large byte counts has always presented problems. Ideas such as \texttt{read()} and \texttt{write()} (using and returning longs) were considered at one time. The current solution is to use abstract types on the ISO C standard function to \texttt{read()} and \texttt{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 \texttt{size_t} also limit portable I/O requests to the same range. This volume of IEEE Std 1003.1-2001 also limits the range further by requiring that the byte count be limited so that a signed return value remains meaningful. Since the return type is also a (signed) abstract type, the byte count can be defined by the implementation to be larger than an int can hold.

The standard developers considered adding atomicity requirements to a pipe or FIFO, but recognized that due to the nature of pipes and FIFOs there could be no guarantee of atomicity of reads of [PIPE_BUF] or any other size that would be an aid to applications portability.

This volume of IEEE Std 1003.1-2001 requires that no action be taken for \texttt{read()} or \texttt{write()} when \texttt{nbyte} is zero. This is not intended to take precedence over detection of errors (such as invalid buffer pointers or file descriptors). This is consistent with the rest of this volume of IEEE Std 1003.1-2001, but the phrasing here could be misread to require detection of the zero case before any other errors. A value of zero is to be considered a correct value, for which the semantics are a no-op.

I/O is intended to be atomic to ordinary files and pipes and FIFOs. Atomic means that all the bytes from a single operation that started out together end up together, without interleaving from other I/O operations. It is a known attribute of terminals that this is not honored, and terminals are explicitly (and implicitly permanently) excepted, making the behavior unspecified. The behavior for other device types is also left unspecified, but the wording is intended to imply that future standards might choose to specify atomicity (or not).

There were recommendations to add format parameters to \texttt{read()} and \texttt{write()} in order to handle networked transfers among heterogeneous file system and base hardware types. Such a facility may be required for support by the OSI presentation of layer services. However, it was determined that this should correspond with similar C-language facilities, and that is beyond the scope of this volume of IEEE Std 1003.1-2001. The concept was suggested to the developers of the ISO C standard for their consideration as a possible area for future work.

In 4.3 BSD, a \texttt{read()} or \texttt{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, \texttt{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 \texttt{select()} to be used before \texttt{read()} in situations (such as keyboard input) where interruption of I/O due to a signal is desired.

The issue of which files or file types are interruptible is considered an implementation design issue. This is often affected primarily by hardware and reliability issues.

There are no references to actions taken following an ‘unrecoverable error’. It is considered beyond the scope of this volume of IEEE Std 1003.1-2001 to describe what happens in the case of hardware errors.

Previous versions of IEEE Std 1003.1-2001 allowed two very different behaviors with regard to the handling of interrupts. In order to minimize the resulting confusion, it was decided that IEEE Std 1003.1-2001 should support only one of these behaviors. Historical practice on AT&T-derived systems was to have \texttt{read()} and \texttt{write()} return -1 and set \texttt{errno} to [EINTR] when interrupted after some, but not all, of the data requested had been transferred. However, the U.S. Department of Commerce FIPS 151-1 and FIPS 151-2 require the historical BSD behavior, in
which `read()` and `write()` return the number of bytes actually transferred before the interrupt. If
−1 is returned when any data is transferred, it is difficult to recover from the error on a seekable
device and impossible on a non-seekable device. Most new implementations support this
behavior. The behavior required by IEEE Std 1003.1-2001 is to return the number of bytes
transferred.

IEEE Std 1003.1-2001 does not specify when an implementation that buffers `read()`s actually
moves the data into the user-supplied buffer, so an implementation may choose to do this at the
latest possible moment. Therefore, an interrupt arriving earlier may not cause `read()` to return a
partial byte count, but rather to return −1 and set `errno` to [EINTR].

Consideration was also given to combining the two previous options, and setting `errno` to
[EINTR] while returning a short count. However, not only is there no existing practice that
implements this, it is also contradictory to the idea that when `errno` is set, the function
responsible shall return −1.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`fcntl()`, `ioctl()`, `lseek()`, `open()`, `pipe()`, `readv()`, the Base Definitions volume of
IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, `<stropts.h>`, `<sys/uio.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.
The `pread()` function is added.

**Issue 6**
The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are
marked as part of the XSI STREAMS Option Group.
The following new requirements on POSIX implementations derive from alignment with the
Single UNIX Specification:
• The DESCRIPTION now states that if `read()` is interrupted by a signal after it has successfully
read some data, it returns the number of bytes read. In Issue 3, it was optional whether `read()`
returned the number of bytes read, or whether it returned −1 with `errno` set to [EINTR]. This
is a FIPS requirement.
• In the DESCRIPTION, text is added to indicate that for regular files, no data transfer occurs
past the offset maximum established in the open file description associated with `fildes`. This
change is to support large files.
• The [EOVERFLOW] mandatory error condition is added.
• The [ENXIO] optional error condition is added.
Text referring to sockets is added to the DESCRIPTION.
The following changes were made to align with the IEEE P1003.1a draft standard:
• The effect of reading zero bytes is clarified.
The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that
read() results are unspecified for typed memory objects.

New RATIONALE is added to explain the atomicity requirements for input and output
operations.

The following error conditions are added for operations on sockets: [EAGAIN], [ECONNRESET], [ENOTCONN], and [ETIMEDOUT].

The [EIO] error is changed to “may fail”.

The following error conditions are added for operations on sockets: [ENOBUFS] and [ENOMEM].

The readv() function is split out into a separate reference page.
NAME
readdir, readdir_r — read a directory

SYNOPSIS
#include <dirent.h>

struct dirent *readdir(DIR *dirp);

TSF
int readdir_r(DIR *restrict dirp, struct dirent *restrict entry, 
struct dirent **restrict result);

DESCRIPTION
The type DIR, which is defined in the <dirent.h> header, represents a directory stream, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files; files may be removed from a directory or added to a directory asynchronously to the operation of readdir().

The readdir() function shall return a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument dirp, and position the directory stream at the next entry. It shall return a null pointer upon reaching the end of the directory stream. The structure dirent defined in the <dirent.h> header describes a directory entry.

The readdir() function shall not return directory entries containing empty names. If entries for dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-dot; otherwise, they shall not be returned.

The pointer returned by readdir() points to data which may be overwritten by another call to readdir() on the same directory stream. This data is not overwritten by another call to readdir() on a different directory stream.

If a file is removed from or added to the directory after the most recent call to opendir() or rewinddir(), whether a subsequent call to readdir() returns an entry for that file is unspecified.

The readdir() function may buffer several directory entries per actual read operation; readdir() shall mark for update the st_atime field of the directory each time the directory is actually read.

After a call to fork(), either the parent or child (but not both) may continue processing the directory stream using readdir(), rewinddir(), or seekdir(). If both the parent and child processes use these functions, the result is undefined.

If the entry names a symbolic link, the value of the d_ino member is unspecified.

The readdir() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The readdir_r() function shall initialize the dirent structure referenced by entry to represent the directory entry at the current position in the directory stream referred to by dirp, store a pointer to this structure at the location referenced by result, and position the directory stream at the next entry.

The storage pointed to by entry shall be large enough for a dirent with an array of char d_name members containing at least {NAME_MAX}+1 elements.

Upon successful return, the pointer returned at *result shall have the same value as the argument entry. Upon reaching the end of the directory stream, this pointer shall have the value NULL.

The readdir_r() function shall not return directory entries containing empty names.
System Interfaces

**readdir()**

If a file is removed from or added to the directory after the most recent call to `opendir()` or `rewinddir()`, whether a subsequent call to `readdir_r()` returns an entry for that file is unspecified.

The `readdir_r()` function may buffer several directory entries per actual read operation; the `readdir_r()` function shall mark for update the `st_atime` field of the directory each time the directory is actually read.

Applications wishing to check for error situations should set `errno` to 0 before calling `readdir()`. If `errno` is set to non-zero on return, an error occurred.

**RETURN VALUE**

Upon successful completion, `readdir()` shall return a pointer to an object of type `struct dirent`. When an error is encountered, a null pointer shall be returned and `errno` shall be set to indicate the error. When the end of the directory is encountered, a null pointer shall be returned and `errno` is not changed.

If successful, the `readdir_r()` function shall return zero; otherwise, an error number shall be returned to indicate the error.

**ERRORS**

The `readdir()` function shall fail if:

- `EOVERFLOW` One of the values in the structure to be returned cannot be represented correctly.

The `readdir()` function may fail if:

- `EBADF` The `dirp` argument does not refer to an open directory stream.
- `ENOENT` The current position of the directory stream is invalid.

The `readdir_r()` function may fail if:

- `EBADF` The `dirp` argument does not refer to an open directory stream.

**EXAMPLES**

The following sample code searches the current directory for the entry `name`.

```c

dirp = opendir(".");
while (dirp) {
    errno = 0;
    if ((dp = readdir(dirp)) != NULL) {
        if (strcmp(dp->d_name, name) == 0) {
            closedir(dirp);
            return FOUND;
        }
    } else {
        if (errno == 0) {
            closedir(dirp);
            return NOT_FOUND;
        }
        closedir(dirp);
        return READ_ERROR;
    }
}
return OPEN_ERROR;
```
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.

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:

```c
int derror (dirp)
DIR *dirp;

void clearderr (dirp)
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.

An error or signal indicating that a directory has changed while open was considered but rejected.

The thread-safe version of the directory reading function returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call. Either the `NAME_MAX` compile-time constant or the corresponding `pathconf()` option can be used to determine the maximum sizes of returned pathnames.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`closedir()`, `lstat()`, `opendir()`, `rewinddir()`, `symlink()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<dirent.h>`, `<sys/types.h>`

**CHANGE HISTORY**

First released in Issue 2.

**Issue 5**

Large File Summit extensions are added.

The `readdir_r()` function is included for alignment with the POSIX Threads Extension.

A note indicating that the `readdir()` function need not be reentrant is added to the DESCRIPTION.
The `readdir_r()` function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U026/7 is applied, correcting the prototype for `readdir_r()`.

The Open Group Corrigendum U026/8 is applied, clarifying the wording of the successful return for the `readdir_r()` function.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

1. 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.

2. A statement is added to the DESCRIPTION indicating the disposition of certain fields in `struct dirent` when an entry refers to a symbolic link.

3. The `[EOVERFLOW]` mandatory error condition is added. This change is to support large files.

4. The `[ENOENT]` optional error condition is added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The `restrict` keyword is added to the `readdir_r()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
readlink — read the contents of a symbolic link

SYNOPSIS
#include <unistd.h>
ssize_t readlink(const char *restrict path, char *restrict buf,
size_t bufsize);

DESCRIPTION
The readlink() function shall place the contents of the symbolic link referred to by path in the
buffer buf which has size bufsize. If the number of bytes in the symbolic link is less than bufsize,
the contents of the remainder of buf are unspecified. If the buf argument is not large enough to
contain the link content, the first bufsize bytes shall be placed in buf.
If the value of bufsize is greater than {SSIZE_MAX}, the result is implementation-defined.

RETURN VALUE
Upon successful completion, readlink() shall return the count of bytes placed in the buffer.
Otherwise, it shall return a value of −1, leave the buffer unchanged, and set errno to indicate the
error.

ERRORS
The readlink() function shall fail if:

[EACCES] Search permission is denied for a component of the path prefix of path.
[EINVAL] The path argument names a file that is not a symbolic link.
[EIO] An I/O error occurred while reading from the file system.
[ELOOP] A loop exists in symbolic links encountered during resolution of the path argument.
[ENAMETOOLONG] The length of the path argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT] A component of path does not name an existing file or path is an empty string.
[ENOTDIR] A component of the path prefix is not a directory.

The readlink() function may fail if:

[EACCES] Read permission is denied for the directory.
[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the path argument.
[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}.
EXAMPLES

Reading the Name of a Symbolic Link

The following example shows how to read the name of a symbolic link named /modules/pass1.

```c
#include <unistd.h>
char buf[1024];
ssizet_t len;
...
if ((len = readlink("/modules/pass1", buf, sizeof(buf)-1)) != -1)
    buf[len] = '\0';
```

APPLICATION USAGE

Conforming applications should not assume that the returned contents of the symbolic link are null-terminated.

RATIONALE

Since IEEE Std 1003.1-2001 does not require any association of file times with symbolic links, there is no requirement that file times be updated by readlink(). The type associated with bufsiz is a size_t in order to be consistent with both the ISO C standard and the definition of read(). The behavior specified for readlink() when bufsiz is zero represents historical practice. For this case, the standard developers considered a change whereby readlink() would return the number of non-null bytes contained in the symbolic link with the buffer buf remaining unchanged; however, since the stat structure member st_size value can be used to determine the size of buffer necessary to contain the contents of the symbolic link as returned by readlink(), this proposal was rejected, and the historical practice retained.

FUTURE DIRECTIONS

None.

SEE ALSO

lstat(), stat(), symlink(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The return type is changed to ssize_t, to align with the IEEE P1003.1a draft standard.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

• This function is made mandatory.

• In this function it is possible for the return value to exceed the range of the type size_t (since size_t has a larger range of positive values than ssize_t). A sentence restricting the size of the size_t object is added to the description to resolve this conflict.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

• The FUTURE DIRECTIONS section is changed to None.

The following changes were made to align with the IEEE P1003.1a draft standard:

• The [ELOOP] optional error condition is added.
The `restrict` keyword is added to the `readlink()` prototype for alignment with the ISO/IEC 9899:1999 standard.
readv()

NAME
readv — read a vector

SYNOPSIS
XSI
#include <sys/uio.h>

ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);

DESCRIPTION
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}.

Each iovec entry specifies the base address and length of an area in memory where data should be placed. The readv() function shall always fill an area completely before proceeding to the next.

Upon successful completion, readv() shall mark for update the st_atime field of the file.

RETURN VALUE
Refer to read().

ERRORS
Refer to read().

In addition, the readv() function shall fail if:

[EINVAL] The sum of the iov_len values in the iov array overflowed an ssize_t.

The readv() function may fail if:

[EINVAL] The iovcnt argument was less than or equal to 0, or greater than [IOV_MAX].

EXAMPLES

Reading Data into an Array
The following example reads data from the file associated with the file descriptor fd into the buffers specified by members of the iov array.

#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
...
ssize_t bytes_read;
int fd;
char buf0[20];
char buf1[30];
char buf2[40];
int iovcnt;
struct iovec iov[3];

iov[0].iov_base = buf0;
iov[0].iov_len = sizeof(buf0);
iov[1].iov_base = buf1;
iov[1].iov_len = sizeof(buf1);
iov[2].iov_base = buf2;
iov[2].iov_len = sizeof(buf2);
readv()

iovcnt = sizeof(iov) / sizeof(struct iovec);

bytes_read = readv(fd, iov, iovcnt);

...  

APPLICATION USAGE
None.

RATIONALE
Refer to read().

FUTURE DIRECTIONS
None.

SEE ALSO
read(), writev(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/uio.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

ISSUE 6
Split out from the read() reference page.
NAME
realloc — memory reallocator

SYNOPSIS
#include <stdlib.h>
void *realloc(void *ptr, size_t size);

DESCRIPTION
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

The realloc() function shall change the size of the memory object pointed to by ptr to the size
specified by size. The contents of the object shall remain unchanged up to the lesser of the new
and old sizes. If the new size of the memory object would require movement of the object, the
space for the previous instantiation of the object is freed. If the new size is larger, the contents of
the newly allocated portion of the object are unspecified. If size is 0 and ptr is not a null pointer,
the object pointed to is freed. If the space cannot be allocated, the object shall remain unchanged.
If ptr is a null pointer, realloc() shall be equivalent to malloc() for the specified size.
If ptr does not match a pointer returned earlier by calloc(), malloc(), or realloc() or if the space has
previously been deallocated by a call to free() or realloc(), the behavior is undefined.

The order and contiguity of storage allocated by successive calls to realloc() 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 shall point to the start (lowest byte address)
of the allocated space. If the space cannot be allocated, a null pointer shall be returned.

RETURN VALUE
Upon successful completion with a size not equal to 0, realloc() shall return a pointer to the
(possibly moved) allocated space. If size is 0, either a null pointer or a unique pointer that can be
successfully passed to free() shall be returned. If there is not enough available memory, realloc() shall return a null pointer and set errno to [ENOMEM].

ERRORS
The realloc() function shall fail if:

[ENOMEM] Insufficient memory is available.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
calloc(), free(), malloc(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>
First released in Issue 1. Derived from Issue 1 of the SVID.

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, if there is not enough available memory, the setting of `errno` to `[ENOMEM]` is added.
- The `[ENOMEM]` error condition is added.
NAME
realpath — resolve a pathname

SYNOPSIS
XSI
#include <stdlib.h>

char *realpath(const char *restrict file_name,
char *restrict resolved_name);

DESCRIPTION
The `realpath()` function shall derive, from the pathname pointed to by `file_name`, an absolute
pathname that names the same file, whose resolution does not involve ".", ".", ".", or symbolic
links. The generated pathname shall be stored as a null-terminated string, up to a maximum of
{PATH_MAX} bytes, in the buffer pointed to by `resolved_name`.

RETURN VALUE
Upon successful completion, `realpath()` shall return a pointer to the resolved name. Otherwise,
`realpath()` shall return a null pointer and set `errno` to indicate the error, and the contents of the
buffer pointed to by `resolved_name` are undefined.

ERRORS
The `realpath()` function shall fail if:

- [EACCES] Read or search permission was denied for a component of `file_name`.
- [EINVAL] Either the `file_name` or `resolved_name` argument is a null pointer.
- [EIO] An error occurred while reading from the file system.
- [ELOOP] A loop exists in symbolic links encountered during resolution of the `path` argument.
- [ENAMETOOLONG] The length of the `file_name` argument exceeds {PATH_MAX} or a pathname
  component is longer than {NAME_MAX}.
- [ENOENT] A component of `file_name` does not name an existing file or `file_name` points to
  an empty string.
- [ENOTDIR] A component of the path prefix is not a directory.

The `realpath()` function may fail if:

- [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
  resolution of the `path` argument.
- [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
  whose length exceeds {PATH_MAX}.
- [ENOMEM] Insufficient storage space is available.
EXAMPLES

Generating an Absolute Pathname

The following example generates an absolute pathname for the file identified by the `symlinkpath` argument. The generated pathname is stored in the `actualpath` array.

```c
#include <stdlib.h>
...
char *symlinkpath = "/tmp/symlink/file";
char actualpath [PATH_MAX+1];
char *ptr;
ptr = realpath(symlinkpath, actualpath);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`getcwd()`, `sysconf()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The `restrict` keyword is added to the `realpath()` prototype for alignment with the ISO/IEC 9899:1999 standard.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.
NAME
recv — receive a message from a connected socket

SYNOPSIS
#include <sys/socket.h>
ssize_t recv(int socket, void *buffer, size_t length, int flags);

DESCRIPTION
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.

The recv() function takes the following arguments:

socket Specifies the socket file descriptor.
buffer Points to a buffer where the message should be stored.
length Specifies the length in bytes of the buffer pointed to by the buffer argument.
flags Specifies the type of message reception. Values of this argument are formed by
logically OR'ing zero or more of the following values:

MSG_PEEK Peeks at an incoming message. The data is treated as unread and
the next recv() or similar function shall still return this data.
MSG_OOB Requests out-of-band data. The significance and semantics of
out-of-band data are protocol-specific.
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.

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.

If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first
message.

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].

RETURN VALUE
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.
The `recv()` function shall fail if:

- [EAGAIN] or [EWOULDBLOCK]
  - The socket’s file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket’s file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.
- [EBADF]
  - The `socket` argument is not a valid file descriptor.
- [ECONNRESET]
  - A connection was forcibly closed by a peer.
- [EINTR]
  - The `recv()` function was interrupted by a signal that was caught, before any data was available.
- [EINVAL]
  - The MSG_OOB flag is set and no out-of-band data is available.
- [ENOTCONN]
  - A receive is attempted on a connection-mode socket that is not connected.
- [ENOTSOCK]
  - The `socket` argument does not refer to a socket.
- [EOPNOTSUPP]
  - The specified flags are not supported for this socket type or protocol.
- [ETIMEDOUT]
  - The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The `recv()` function may fail if:

- [EIO]
  - An I/O error occurred while reading from or writing to the file system.
- [ENOBUFS]
  - Insufficient resources were available in the system to perform the operation.
- [ENOMEM]
  - Insufficient memory was available to fulfill the request.

**EXAMPLES**

None.

**APPLICATION USAGE**

The `recv()` function is equivalent to `recvfrom()` with a zero `address_len` argument, and to `read()` if no flags are used.

The `select()` and `poll()` functions can be used to determine when data is available to be received.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`poll()`, `read()`, `recvmsg()`, `recvfrom()`, `select()`, `send()`, `sendmsg()`, `sendto()`, `shutdown()`, `socket()`, `write()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/socket.h>`

**CHANGE HISTORY**

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
recvfrom — receive a message from a socket

SYNOPSIS
#include <sys/socket.h>
ssize_t recvfrom(int socket, void *restrict buffer, size_t length,
                  int flags, struct sockaddr *restrict address,
                  socklen_t *restrict address_len);

DESCRIPTION
The recvfrom() 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.

The recvfrom() function takes the following arguments:

socket Specifies the socket file descriptor.
buffer Points to the buffer where the message should be stored.
length Specifies the length in bytes of the buffer pointed to by the buffer argument.
flags Specifies the type of message reception. Values of this argument are formed
by logically OR’ing zero or more of the following values:
    MSG_PEEK Peeks at an incoming message. The data is treated as unread
    and the next recvfrom() or similar function shall still return
    this data.
    MSG_OOB Requests out-of-band data. The significance and semantics
    of out-of-band data are protocol-specific.
    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.
address A null pointer, or points to a sockaddr structure in which the sending address
is to be stored. The length and format of the address depend on the address
family of the socket.
address_len Specifies the length of the sockaddr structure pointed to by the address
argument.

The recvfrom() 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_RAW, 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.

If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first
message.

Not all protocols provide the source address for messages. If the address argument is not a null
pointer and the protocol provides the source address of messages, the source address of the
received message shall be stored in the `sockaddr` structure pointed to by the `address` argument, and the length of this address shall be stored in the object pointed to by the `address_len` argument.

If the actual length of the address is greater than the length of the supplied `sockaddr` structure, the stored address shall be truncated.

If the `address` argument is not a null pointer and the protocol does not provide the source address of messages, the value stored in the object pointed to by `address` is unspecified.

If no messages are available at the socket and O_NONBLOCK is not set on the socket’s file descriptor, `recvfrom()` 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, `recvfrom()` shall fail and set `errno` to [EAGAIN] or [EWOULDBLOCK].

**RETURN VALUE**
Upon successful completion, `recvfrom()` 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, `recvfrom()` shall return 0. Otherwise, the function shall return -1 and set `errno` to indicate the error.

**ERRORS**
The `recvfrom()` function shall fail if:

- [EAGAIN] or [EWOULDBLOCK]
  - The socket’s file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket’s file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.
- [EBADF]
  - The `socket` argument is not a valid file descriptor.
- [ECONNRESET]
  - A connection was forcibly closed by a peer.
- [EINTR]
  - A signal interrupted `recvfrom()` before any data was available.
- [EINVAL]
  - The MSG_OOB flag is set and no out-of-band data is available.
- [ENOTCONN]
  - A receive is attempted on a connection-mode socket that is not connected.
- [ENOTSOCK]
  - The `socket` argument does not refer to a socket.
- [EOPNOTSUPP]
  - The specified flags are not supported for this socket type.
- [ETIMEDOUT]
  - The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The `recvfrom()` function may fail if:

- [EIO]
  - An I/O error occurred while reading from or writing to the file system.
- [ENOBUFS]
  - Insufficient resources were available in the system to perform the operation.
- [ENOMEM]
  - Insufficient memory was available to fulfill the request.
EXAMPLES
None.

APPLICATION USAGE
The select() and poll() functions can be used to determine when data is available to be received.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
poll(), read(), recv(), recvmsg(), select(), send(), sendmsg(), sendto(), shutdown(), socket(), write(),
the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
recvmsg — receive a message from a socket

SYNOPSIS
#include <sys/socket.h>
ssize_t recvmsg(int socket, struct msghdr *message, int flags);

DESCRIPTION
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.
The recvmsg() function takes the following arguments:

socket Specifies the socket file descriptor.
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.
flags Specifies the type of message reception. Values of this argument are formed
by logically OR'ing zero or more of the following values:

MSG_OOB Requests out-of-band data. The significance and semantics
of out-of-band data are protocol-specific.
MSG_PEEK Peeks at the incoming message.
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.
The recvmsg() function shall receive messages from unconnected or connected sockets and shall
return the length of the message.
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.
If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first
message.
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].
In the msghdr structure, the msg_name and msg_name len members specify the source address if
the socket is unconnected. If the socket is connected, the msg_name and msg_name len 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_iovlen fields are used to specify where the received data shall be
recvmsg()

System Interfaces

37120 stored. msg iov points to an array of iovec structures; msg iovlen shall be set to the dimension of
37121 this array. In each iovec structure, the iov base field specifies a storage area and the iov len field
37122 gives its size in bytes. Each storage area indicated by msg iov is filled with received data in turn
37123 until all of the received data is stored or all of the areas have been filled.
37124 Upon successful completion, the msg_flags member of the message header shall be the bitwise-
37125 inclusive OR of all of the following flags that indicate conditions detected for the received
37126 message:
37127 MSG_EOR End-of-record was received (if supported by the protocol).
37128 MSG_OOB Out-of-band data was received.
37129 MSG_TRUNC Normal data was truncated.
37130 MSG_CTRUNC Control data was truncated.
37131 RETURN VALUE
37132 Upon successful completion, recvmsg() shall return the length of the message in bytes. If no
37133 messages are available to be received and the peer has performed an orderly shutdown,
37134 recvmsg() shall return 0. Otherwise, −1 shall be returned and errno set to indicate the error.
37135 ERRORS
37136 The recvmsg() function shall fail if:
37137 [EAGAIN] or [EWOULDBLOCK]
37138 The socket’s file descriptor is marked O_NONBLOCK and no data is waiting
37139 to be received; or MSG_OOB is set and no out-of-band data is available and
37140 either the socket’s file descriptor is marked O_NONBLOCK or the socket does
37141 not support blocking to await out-of-band data.
37142 [EBADF] The socket argument is not a valid open file descriptor.
37143 [ECONNRESET] A connection was forcibly closed by a peer.
37144 [EINTR] This function was interrupted by a signal before any data was available.
37145 [EINVAL] The sum of the iov len values overflows a ssize_t, or the MSG_OOB flag is set
37146 and no out-of-band data is available.
37147 [EMSGSIZE] The msg iovlen member of the msghdr structure pointed to by message is less
37148 than or equal to 0, or is greater than [IOV_MAX].
37149 [ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.
37150 [ENOTSOCK] The socket argument does not refer to a socket.
37151 [EOPNOTSUPP] The specified flags are not supported for this socket type.
37152 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
37153 transmission timeout on active connection.
37154 The recvmsg() function may fail if:
37155 [EIO] An I/O error occurred while reading from or writing to the file system.
37156 [ENOBUFFS] Insufficient resources were available in the system to perform the operation.
37157 [ENOMEM] Insufficient memory was available to fulfill the request.
EXAMPLES
None.

APPLICATION USAGE
The select() and poll() functions can be used to determine when data is available to be received.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
poll(), recv(), recvfrom(), select(), send(), sendmsg(), sendto(), shutdown(), socket(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
regcomp, regerror, regex, regfree — regular expression matching

SYNOPSIS
#include <regex.h>

int regcomp(regex_t *restrict preg, const char *restrict pattern,
            int cflags);
size_t regerror(int errcode, const regex_t *restrict preg,
                 char *restrict errbuf, size_t errbuf_size);
int regex(const regex_t *restrict preg, const char *restrict string,
          size_t nmatch, regmatch_t pmatch[restrict], int eflags);
void regfree(regex_t *preg);

DESCRIPTION
These functions interpret basic and extended regular expressions as described in the Base
The regex_t structure is defined in <regex.h> and contains at least the following member:

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t</td>
<td>re_nsub</td>
<td>Number of parenthesized subexpressions.</td>
</tr>
</tbody>
</table>

The regmatch_t structure is defined in <regex.h> and contains at least the following members:

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>regoff_t</td>
<td>rm_so</td>
<td>Byte offset from start of string to start of substring.</td>
</tr>
<tr>
<td>regoff_t</td>
<td>rm_eo</td>
<td>Byte offset from start of string of the first character after the end of substring.</td>
</tr>
</tbody>
</table>

The regcomp() function shall compile the regular expression contained in the string pointed to by
the pattern argument and place the results in the structure pointed to by preg. The cflags
argument is the bitwise-inclusive OR of zero or more of the following flags, which are defined in
the <regex.h> header:

REG_EXTENDED Use Extended Regular Expressions.
REG_ICASE Ignore case in match. (See the Base Definitions volume of
IEEE Std 1003.1-2001, Chapter 9, Regular Expressions.)
REG_NOSUB Report only success/fail in regex().
REG_NEWLINE Change the handling of <newline>s, as described in the text.

The default regular expression type for pattern is a Basic Regular Expression. The application can
specify Extended Regular Expressions using the REG_EXTENDED cflags flag.

If the REG_NOSUB flag was not set in cflags, then regcomp() shall set re_nsub to the number of
parenthesized subexpressions (delimited by "\(\)" in basic regular expressions or "\(\)" in
extended regular expressions) found in pattern.

The regex() function compares the null-terminated string specified by string with the compiled
regular expression preg initialized by a previous call to regcomp(). If it finds a match, regex() shall return 0; otherwise, it shall return non-zero indicating either no match or an error. The
eflags argument is the bitwise-inclusive OR of zero or more of the following flags, which are
defined in the <regex.h> header:
REG_NOTBOL The first character of the string pointed to by string is not the beginning of the line. Therefore, the circumflex character (‘ˆ’), when taken as a special character, shall not match the beginning of string.

REG_NOTEOL The last character of the string pointed to by string is not the end of the line. Therefore, the dollar sign (‘$’), when taken as a special character, shall not match the end of string.

If nmatch is 0 or REG_NOSUB was set in the cflags argument to regcomp(), then regexec() shall ignore the pmatch argument. Otherwise, the application shall ensure that the pmatch argument points to an array with at least nmatch elements, and regexec() shall fill in the elements of that array with offsets of the substrings of string that correspond to the parenthesized subexpressions of pattern: pmatch[i].rm_so shall be the byte offset of the beginning and pmatch[i].rm_eo shall be one greater than the byte offset of the end of substring i. (Subexpression i begins at the ith matched open parenthesis, counting from 1.) Offsets in pmatch[0] identify the substring that corresponds to the entire regular expression. Unused elements of pmatch up to pmatch[nmatch−1] shall be filled with −1. If there are more than nmatch subexpressions in pattern (pattern itself counts as a subexpression), then regexec() shall still do the match, but shall record only the first nmatch substrings.

When matching a basic or extended regular expression, any given parenthesized subexpression of pattern might participate in the match of several different substrings of string, or it might not match any substring even though the pattern as a whole did match. The following rules shall be used to determine which substrings to report in pmatch when matching regular expressions:

1. If subexpression i in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in pmatch[i] shall delimit the last such match.

2. If subexpression i is not contained within another subexpression, and it did not participate in an otherwise successful match, the byte offsets in pmatch[i] shall be −1. A subexpression does not participate in the match when:
   `'*'` or `"\(\)"` appears immediately after the subexpression in a basic regular expression, or `'*', '?'`, or `"()"` appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched 0 times) or:
   `' |` is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.

3. If subexpression i is contained within another subexpression j, and i is not contained within any other subexpression that is contained within j, and a match of subexpression j is reported in pmatch[j], then the match or non-match of subexpression i reported in pmatch[i] shall be as described in 1. and 2. above, but within the substring reported in pmatch[j] rather than the whole string. The offsets in pmatch[i] are still relative to the start of string.

4. If subexpression i is contained in subexpression j, and the byte offsets in pmatch[j] are −1, then the pointers in pmatch[i] shall also be −1.

5. If subexpression i matched a zero-length string, then both byte offsets in pmatch[i] shall be the byte offset of the character or null terminator immediately following the zero-length string.

If, when regexec() is called, the locale is different from when the regular expression was compiled, the result is undefined.
If REG_NEWLINE is not set in cflags, then a <newline> in pattern or string shall be treated as an ordinary character. If REG_NEWLINE is set, then <newline> shall be treated as an ordinary character except as follows:

1. A <newline> in string shall not be matched by a period outside a bracket expression or by any form of a non-matching list (see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 9, Regular Expressions).

2. A circumflex (’ˆ’) in pattern, when used to specify expression anchoring (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 9.3.8, BRE Expression Anchoring), shall match the zero-length string immediately after a <newline> in string, regardless of the setting of REG_NOTBOL.

3. A dollar sign (’$’) in pattern, when used to specify expression anchoring, shall match the zero-length string immediately before a <newline> in string, regardless of the setting of REG_NOTEOL.

The regfree() function frees any memory allocated by regcomp() associated with preg.

The following constants are defined as error return values:

- REG_NOMATCH: regexec() failed to match.
- REG_BADPAT: Invalid regular expression.
- REG_ECOLLATE: Invalid collating element referenced.
- REG_ECTYPE: Invalid character class type referenced.
- REG_EESCAPE: Trailing ’\’ in pattern.
- REG_ESUBREG: Number in "\digit" invalid or in error.
- REG_EBRACK: "[ ]" imbalance.
- REG_EPAREN: "\(\)" or "()" imbalance.
- REG_EBRACE: "\(\)" imbalance.
- REG_BADBR: Content of "\(\)" invalid: not a number, number too large, more than two numbers, first larger than second.
- REG_ERANGE: Invalid endpoint in range expression.
- REG_ESPACE: Out of memory.
- REG_BADRPT: ’?’, ’*’, or ’+’ not preceded by valid regular expression.

The regerror() function provides a mapping from error codes returned by regcomp() and regexec() to unspecified printable strings. It generates a string corresponding to the value of the errcode argument, which the application shall ensure is the last non-zero value returned by regcomp() or regexec() with the given value of preg. If errcode is not such a value, the content of the generated string is unspecified.

If preg is a null pointer, but errcode is a value returned by a previous call to regexec() or regcomp(), the regerror() still generates an error string corresponding to the value of errcode, but it might not be as detailed under some implementations.

If the errbuf_size argument is not 0, regerror() shall place the generated string into the buffer of size errbuf_size bytes pointed to by errbuf. If the string (including the terminating null) cannot fit in the buffer, regerror() shall truncate the string and null-terminate the result.
If `errbuf_size` is 0, `regerror()` shall ignore the `errbuf` argument, and return the size of the buffer needed to hold the generated string.

If the `preg` argument to `regexec()` or `regfree()` is not a compiled regular expression returned by `regcomp()`, the result is undefined. A `preg` is no longer treated as a compiled regular expression after it is given to `regfree()`.

**RETURN VALUE**

Upon successful completion, the `regcomp()` function shall return 0. Otherwise, it shall return an integer value indicating an error as described in `<regex.h>`, and the content of `preg` is undefined.

If a code is returned, the interpretation shall be as given in `<regex.h>`.

If `regcomp()` detects an invalid RE, it may return `REG_BADPAT`, or it may return one of the error codes that more precisely describes the error.

Upon successful completion, the `regexec()` function shall return 0. Otherwise, it shall return `REG_NOMATCH` to indicate no match.

Upon successful completion, the `regerror()` function shall return the number of bytes needed to hold the entire generated string, including the null termination. If the return value is greater than `errbuf_size`, the string returned in the buffer pointed to by `errbuf` has been truncated.

The `regfree()` function shall not return a value.

**ERRORS**

No errors are defined.

**EXAMPLES**

```c
#include <regex.h>

/*
 * Match string against the extended regular expression in
 * pattern, treating errors as no match.
 * Return 1 for match, 0 for no match.
 */

int match(const char *string, char *pattern)
{
    int status;
    regex_t re;
    if (regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB) != 0) {
        return(0); /* Report error. */
    }
    status = regexec(&re, string, (size_t) 0, NULL, 0);
    regfree(&re);
    if (status != 0) {
        return(0); /* Report error. */
    }
    return(1);
}
```

The following demonstrates how the REG_NOTBOL flag could be used with `regexec()` to find all substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very little error checking is done.)
(void) regcomp (&re, pattern, 0);
/* This call to regexec() finds the first match on the line. */
error = regexec (&re, &buffer[0], 1, &pm, 0);
while (error == 0) { /* While matches found. */
    /* Substring found between pm.rm_so and pm.rm_eo. */
    /* This call to regexec() finds the next match. */
    error = regexec (&re, buffer + pm.rm_eo, 1, &pm, REG_NOTBOL);
}

APPLICATION USAGE
An application could use:
regerror(code,preg,(char *)NULL,(size_t)0)
to find out how big a buffer is needed for the generated string, malloc() a buffer to hold the
string, and then call regerror() again to get the string. Alternatively, it could allocate a fixed,
static buffer that is big enough to hold most strings, and then use malloc() to allocate a larger
buffer if it finds that this is too small.
To match a pattern as described in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section
2.13, Pattern Matching Notation, use the fnmatch() function.

RATIONALE
The regexec() function must fill in all nmatch elements of pmatch, where nmatch and pmatch are
supplied by the application, even if some elements of pmatch do not correspond to
subexpressions in pattern. The application writer should note that there is probably no reason
for using a value of nmatch that is larger than preg->re_nsub+1.
The REG_NEWLINE flag supports a use of RE matching that is needed in some applications like
text editors. In such applications, the user supplies an RE asking the application to find a line
that matches the given expression. An anchor in such an RE anchors at the beginning or end of
any line. Such an application can pass a sequence of <newline>-separated lines to regexec() as a
single long string and specify REG_NEWLINE to regcomp() to get the desired behavior. The
application must ensure that there are no explicit <newline>s in pattern if it wants to ensure that
any match occurs entirely within a single line.
The REG_NEWLINE flag affects the behavior of regexec(), but it is in the cflags parameter to
regcomp() to allow flexibility of implementation. Some implementations will want to generate
the same compiled RE in regcomp() regardless of the setting of REG_NEWLINE and have
regexec() handle anchors differently based on the setting of the flag. Other implementations will
generate different compiled REs based on the REG_NEWLINE.
The REG_ICASE flag supports the operations taken by the grep −i option and the historical
implementations of ex and vi. Including this flag will make it easier for application code to be
written that does the same thing as these utilities.
The substrings reported in pmatch[] are defined using offsets from the start of the string rather
than pointers. Since this is a new interface, there should be no impact on historical
implementations or applications, and offsets should be just as easy to use as pointers. The
change to offsets was made to facilitate future extensions in which the string to be searched is
presented to regexec() in blocks, allowing a string to be searched that is not all in memory at
once.
The type regoff_t is used for the elements of pmatch[] to ensure that the application can
represent either the largest possible array in memory (important for an application conforming
to the Shell and Utilities volume of IEEE Std 1003.1-2001) or the largest possible file (important
for an application using the extension where a file is searched in chunks).
The standard developers rejected the inclusion of a `regsub()` function that would be used to do
substitutions for a matched RE. While such a routine would be useful to some applications, its
utility would be much more limited than the matching function described here. Both RE parsing
and substitution are possible to implement without support other than that required by the
ISO C standard, but matching is much more complex than substituting. The only difficult part of
substitution, given the information supplied by `regexec()`, is finding the next character in a string
when there can be multi-byte characters. That is a much larger issue, and one that needs a more
general solution.

The `errno` variable has not been used for error returns to avoid filling the `errno` name space for
this feature.

The interface is defined so that the matched substrings `rm_sp` and `rm_ep` are in a separate
`regmatch_t` structure instead of in `regex_t`. This allows a single compiled RE to be used
simultaneously in several contexts; in `main()` and a signal handler, perhaps, or in multiple
threads of lightweight processes. (The `preg` argument to `regexec()` is declared with type `const`, so
the implementation is not permitted to use the structure to store intermediate results.) It also
allows an application to request an arbitrary number of substrings from an RE. The number of
subexpressions in the RE is reported in `re_nsub` in `preg`. With this change to `regexec()`,
consideration was given to dropping the REG_NOSUB flag since the user can now specify this
with a zero `nmatch` argument to `regexec()`. However, keeping REG_NOSUB allows an
implementation to use a different (perhaps more efficient) algorithm if it knows in `regcomp()`
that no subexpressions need be reported. The implementation is only required to fill in `pmatch` if
`nmatch` is not zero and if REG_NOSUB is not specified. Note that the `size_t` type, as defined in
the ISO C standard, is unsigned, so the description of `regexec()` does not need to address
negative values of `nmatch`.

REG_NOTBOL was added to allow an application to do repeated searches for the same pattern
in a line. If the pattern contains a circumflex character that should match the beginning of a line,
then the pattern should only match when matched against the beginning of the line. Without
the REG_NOTBOL flag, the application could rewrite the expression for subsequent matches,
but in the general case this would require parsing the expression. The need for REG_NOTEOL is
not as clear; it was added for symmetry.

The addition of the `regerror()` function addresses the historical need for conforming application
programs to have access to error information more than “Function failed to compile/match your
RE for unknown reasons”.

This interface provides for two different methods of dealing with error conditions. The specific
error codes (REG_EBRACE, for example), defined in `<regex.h>`, allow an application to recover
from an error if it is so able. Many applications, especially those that use patterns supplied by a
user, will not try to deal with specific error cases, but will just use `regerror()` to obtain a human-
readable error message to present to the user.

The `regerror()` function uses a scheme similar to `confstr()` to deal with the problem of allocating
memory to hold the generated string. The scheme used by `strerror()` in the ISO C standard was
considered unacceptable since it creates difficulties for multi-threaded applications.

The `preg` argument is provided to `regerror()` to allow an implementation to generate a more
descriptive message than would be possible with `errcode` alone. An implementation might, for
example, save the character offset of the offending character of the pattern in a field of `preg`, and
then include that in the generated message string. The implementation may also ignore `preg`.

A REG_FILENAME flag was considered, but omitted. This flag caused `regexec()` to match
patterns as described in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.13,
Pattern Matching Notation instead of REs. This service is now provided by the `fnmatch()`
function.

Notice that there is a difference in philosophy between the ISO POSIX-2:1993 standard and IEEE Std 1003.1-2001 in how to handle a “bad” regular expression. The ISO POSIX-2:1993 standard says that many bad constructs “produce undefined results”, or that “the interpretation is undefined”. IEEE Std 1003.1-2001, however, says that the interpretation of such REs is unspecified. The term “undefined” means that the action by the application is an error, of similar severity to passing a bad pointer to a function.

The `regcomp()` and `regexec()` functions are required to accept any null-terminated string as the pattern argument. If the meaning of the string is “undefined”, the behavior of the function is “unspecified”. IEEE Std 1003.1-2001 does not specify how the functions will interpret the pattern; they might return error codes, or they might do pattern matching in some completely unexpected way, but they should not do something like abort the process.

FUTURE DIRECTIONS

None.

SEE ALSO

`fnmatch()`, `glob()`, Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.13, Pattern Matching Notation, Base Definitions volume of IEEE Std 1003.1-2001, Chapter 9, Regular Expressions, `<regex.h>`, `<sys/types.h>`

CHANGE HISTORY


Issue 5

Moved from POSIX2 C-language Binding to BASE.

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.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The REG_ENOSYS constant is removed.

The `restrict` keyword is added to the `regcomp()`, `regerror()`, and `regexec()` prototypes for alignment with the ISO/IEC 9899:1999 standard.
NAME
remainder, remainderf, remainderl — remainder function

SYNOPSIS
#include <math.h>

double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);

DESCRIPTION
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

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 \( |n-x/y| = \frac{1}{2} \), the value \( n \) is chosen to
be even.

The behavior of \( \text{remainder()} \) shall be independent of the rounding mode.

RETURN VALUE
Upon successful completion, these functions shall return the floating-point remainder \( r = x - ny \)
when \( y \) is non-zero.

If \( x \) or \( y \) is NaN, a NaN shall be returned.
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.

ERRORS
These functions shall fail if:

- Domain Error The \( x \) argument is ±Inf, or the \( y \) argument is ±0 and the other argument is
  non-NaN.
- If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then \( \text{errno} \) 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.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
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,
CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The remainder() function is no longer marked as an extension.

The remainderf() and remainderl() functions are added for alignment with the ISO/IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899: 1999 standard.

remove()  

NAME
remove — remove a file

SYNOPSIS
#include <stdio.h>
int remove(const char *path);

DESCRIPTION
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 remove() function shall cause the file named by the pathname pointed to by path to be no longer accessible by that name. A subsequent attempt to open that file using that name shall fail, unless it is created anew.

If path does not name a directory, remove(path) shall be equivalent to unlink(path).
If path names a directory, remove(path) shall be equivalent to rmdir(path).

RETURN VALUE
Refer to rmdir() or unlink().

ERRORS
Refer to rmdir() or unlink().

EXAMPLES

Removing Access to a File
The following example shows how to remove access to a file named /home/cnd/old Mods.

#include <stdio.h>
int status;
...  
status = remove("/home/cnd/old Mods");

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
rmdir(), unlink(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY

Issue 6
Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:
• 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()`.
remque() — remove an element from a queue

SYNOPSIS

#include <search.h>

void remque(void *element);

DESCRIPTION

Refer to insque().
NAME
remquo, remquof, remquol — remainder functions

SYNOPSIS
#include <math.h>

double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y, int *quo);

DESCRIPTION
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.

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.

RETURN VALUE
These functions shall return $x \text{ REM } y$.

If x or y is NaN, a NaN shall be returned.

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.

ERRORS
These functions shall fail if:

Domain Error  The x argument is ±Inf, or the y argument is ±0 and the other argument is
non-NaN.

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.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
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.
remquo()

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept(), fetestexcept(), remainder(), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY

NAME
rename — rename a file

SYNOPSIS
#include <stdio.h>
int rename(const char *old, const char *new);

DESCRIPTION
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 rename() function shall change the name of a file. The old argument points to the pathname of the file to be renamed. The new argument points to the new pathname of the file.

If either the old or new argument names a symbolic link, rename() shall operate on the symbolic link itself, and shall not resolve the last component of the argument. If the old argument and the new argument resolve to the same existing file, rename() shall return successfully and perform no other action.

If the old argument points to the pathname of a file that is not a directory, the new argument shall not point to the pathname of a directory. If the link named by the new argument exists, it shall be removed and old renamed to new. In this case, a link named new shall remain visible to other processes throughout the renaming operation and refer either to the file referred to by new or old before the operation began. Write access permission is required for both the directory containing old and the directory containing new.

If the old argument points to the pathname of a directory, the new argument shall not point to the pathname of a file that is not a directory. If the directory named by the new argument exists, it shall be removed and old renamed to new. In this case, a link named new shall exist throughout the renaming operation and shall refer either to the directory referred to by new or old before the operation began. If new names an existing directory, it shall be required to be an empty directory.

If the old argument points to a pathname of a symbolic link, the symbolic link shall be renamed. If the new argument points to a pathname of a symbolic link, the symbolic link shall be removed.

The new pathname shall not contain a path prefix that names old. Write access permission is required for the directory containing old and the directory containing new. If the old argument points to the pathname of a directory, write access permission may be required for the directory named by old, and, if it exists, the directory named by new.

If the link named by the new argument exists and the file's link count becomes 0 when it is removed and no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible. If one or more processes have the file open when the last link is removed, the link shall be removed before rename() returns, but the removal of the file contents shall be postponed until all references to the file are closed.

Upon successful completion, rename() shall mark for update the st_ctime and st_mtime fields of the parent directory of each file.

If the rename() function fails for any reason other than [EIO], any file named by new shall be unaffected.

RETURN VALUE
Upon successful completion, rename() shall return 0; otherwise, −1 shall be returned, errno shall be set to indicate the error, and neither the file named by old nor the file named by new shall be changed or created.
The `rename()` function shall fail if:

- **[EACCES]** A component of either path prefix denies search permission; or one of the directories containing `old` or `new` denies write permissions; or, write permission is required and is denied for a directory pointed to by the `old` or `new` arguments.
- **[EBUSY]** The directory named by `old` or `new` is currently in use by the system or another process, and the implementation considers this an error.
- **[EEXIST]** or **[ENOTEMPTY]** The link named by `new` is a directory that is not an empty directory.
- **[EINVAL]** The `new` directory pathname contains a path prefix that names the `old` directory.
- **[EIO]** A physical I/O error has occurred.
- **[ELOOP]** A loop exists in symbolic links encountered during resolution of the `path` argument.
- **[EMLINK]** The file named by `old` is a directory, and the link count of the parent directory of `new` would exceed `{LINK_MAX}`.
- **[ENAMETOOLONG]** The length of the `old` or `new` argument exceeds `{PATH_MAX}` or a pathname component is longer than `{NAME_MAX}`.
- **[ENOENT]** The link named by `old` does not name an existing file, or either `old` or `new` points to an empty string.
- **[ENOSPC]** The directory that would contain `new` cannot be extended.
- **[ENOTDIR]** A component of either path prefix is not a directory; or the `old` argument names a directory and `new` argument names a non-directory file.
- **[EPERM]** or **[EACCES]** The S_ISVTX flag is set on the directory containing the file referred to by `old` and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges; or `new` 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.
- **[EROFS]** The requested operation requires writing in a directory on a read-only file system.
- **[EXDEV]** The links named by `new` and `old` are on different file systems and the implementation does not support links between file systems.

The `rename()` function may fail if:

- **[EBUSY]** The file named by the `old` or `new` arguments is a named STREAM.
- **[ELOOP]** More than `{SYMLOOP_MAX}` symbolic links were encountered during resolution of the `path` argument.
System Interfaces

rename()

As a result of encountering a symbolic link in resolution of the path argument, the length of the substituted pathname string exceeded [PATH_MAX].

The file to be renamed is a pure procedure (shared text) file that is being executed.

EXAMPLES

Renaming a File

The following example shows how to rename a file named /home/cnd/mod1 to /home/cnd/mod2.

```
#include <stdio.h>

int status;
...
status = rename("/home/cnd/mod1", "/home/cnd/mod2");
```

APPLICATION USAGE

Some implementations mark for update the st_ctime field of renamed files and some do not. Applications which make use of the st_ctime field may behave differently with respect to renamed files unless they are designed to allow for either behavior.

RATIONALE

This rename() function is equivalent for regular files to that defined by the ISO C standard. Its inclusion here expands that definition to include actions on directories and specifies behavior when the new parameter names a file that already exists. That specification requires that the action of the function be atomic.

One of the reasons for introducing this function was to have a means of renaming directories while permitting implementations to prohibit the use of link() and unlink() with directories, thus constraining links to directories to those made by mkdir().

The specification that if old and new refer to the same file is intended to guarantee that:

```
rename("x", "x");
```

does not remove the file.

Renaming dot or dot-dot is prohibited in order to prevent cyclical file system paths.

See also the descriptions of [ENOTEMPTY] and [ENAMETOOLONG] in rmdir() and [EBUSY] in unlink(). For a discussion of [EXDEV], see link().

FUTURE DIRECTIONS

None.

SEE ALSO

link(), rmdir(), symlink(), unlink(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 5

The [EBUSY] error is added to the “may fail” part of the ERRORS section.
Extensions beyond the ISO C standard are marked.
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

• The [EIO] mandatory error condition is added.
• The [ELOOP] mandatory error condition is added.
• A second [ENAMETOOLONG] is added as an optional error condition.
• The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

• Details are added regarding the treatment of symbolic links.
• The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
rewind — reset the file position indicator in a stream

SYNOPSIS
#include <stdio.h>

void rewind(FILE *stream);

DESCRIPTION
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 call:
rewind(stream)
shall be equivalent to:
(void) fseek(stream, 0L, SEEK_SET)
except that rewind() shall also clear the error indicator.

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.

RETURN VALUE
The rewind() function shall not return a value.

ERRORS
Refer to fseek() with the exception of [EINVAL] which does not apply.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fseek(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.
rewinddir()

NAME
rewinddir — reset the position of a directory stream to the beginning of a directory

SYNOPSIS
#include <dirent.h>

void rewinddir(DIR *dirp);

DESCRIPTION
The rewinddir() function shall reset the position of the directory stream to which dirp refers to
the beginning of the directory. It shall also cause the directory stream to refer to the current state
of the corresponding directory, as a call to opendir() would have done. If dirp does not refer to a
directory stream, the effect is undefined.

After a call to the fork() function, either the parent or child (but not both) may continue
processing the directory stream using readdir(), rewinddir(), or seekdir(). If both the parent and
child processes use these functions, the result is undefined.

RETURN VALUE
The rewinddir() function shall not return a value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
The rewinddir() function should be used in conjunction with opendir(), readdir(), and closedir() to
examine the contents of the directory. This method is recommended for portability.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
closedir(), opendir(), readdir(), the Base Definitions volume of IEEE Std 1003.1-2001, <dirent.h>
<sys/types.h>

CHANGE HISTORY
First released in Issue 2.

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.
NAME
rindex — character string operations (LEGACY)

SYNOPSIS
#include <strings.h>

char *rindex(const char *s, int c);

DESCRIPTION
The rindex() function shall be equivalent to strrchr().

RETURN VALUE
Refer to strrchr().

ERRORS
Refer to strrchr().

EXAMPLES
None.

APPLICATION USAGE
The strrchr() function is preferred over this function.
For maximum portability, it is recommended to replace the function call to rindex() as follows:
#define rindex(a,b) strrchr((a),(b))

RATIONALE
None.

FUTURE DIRECTIONS
This function may be withdrawn in a future version.

SEE ALSO
strrchr(), the Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
This function is marked LEGACY.
NAME
rint, rintf, rintl — round-to-nearest integral value

SYNOPSIS
#include <math.h>

double rint(double x);
float rintf(float x);
long double rintl(long double x);

DESCRIPTION
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

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.

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().

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.

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.

RETURN VALUE
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.

If x is NaN, a NaN shall be returned.

If x is ±0 or ±Inf, x shall be returned.

If the correct value would cause overflow, a range error shall occur and rint(), rintf(), and rintl() shall return the value of the macro ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (with the
same sign as x), respectively.

ERRORS
These functions shall fail if:

Range Error The result would cause an overflow.

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.
EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
abs(), ceil(), feclearexcept(), fetestexcept(), floor(), isnan(), nearbyint(), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

• The rintf() and rintl() functions are added.

• The rint() function is no longer marked as an extension.

• The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

NAME
  rmdir — remove a directory

SYNOPSIS
  #include <unistd.h>
  int rmdir(const char *path);

DESCRIPTION
  The rmdir() function shall remove a directory whose name is given by path. The directory shall be removed only if it is an empty directory.

  If the directory is the root directory or the current working directory of any process, it is unspecified whether the function succeeds, or whether it shall fail and set errno to [EBUSY].

  If path names a symbolic link, then rmdir() shall fail and set errno to [ENOTDIR].

  If the path argument refers to a path whose final component is either dot or dot-dot, rmdir() shall fail.

  If the directory’s link count becomes 0 and no process has the directory open, the space occupied by the directory shall be freed and the directory shall no longer be accessible. If one or more processes have the directory open when the last link is removed, the dot and dot-dot entries, if present, shall be removed before rmdir() returns and no new entries may be created in the directory, but the directory shall not be removed until all references to the directory are closed.

  If the directory is not an empty directory, rmdir() shall fail and set errno to [EEXIST] or [ENOTEMPTY].

  Upon successful completion, the rmdir() function shall mark for update the st_ctime and st_mtime fields of the parent directory.

RETURN VALUE
  Upon successful completion, the function rmdir() shall return 0. Otherwise, −1 shall be returned, and errno set to indicate the error. If −1 is returned, the named directory shall not be changed.

ERRORS
  The rmdir() function shall fail if:

  [EACCES]  Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be removed.

  [EBUSY]    The directory to be removed is currently in use by the system or some process and the implementation considers this to be an error.

  [EEXIST] or [ENOTEMPTY]
  The path argument names a directory that is not an empty directory, or there are hard links to the directory other than dot or a single entry in dot-dot.

  [EINVAL]   The path argument contains a last component that is dot.

  [EIO]      A physical I/O error has occurred.

  [ELOOP]    A loop exists in symbolic links encountered during resolution of the path argument.

  [ENAMETOOLONG]
  The length of the path argument exceeds [PATH_MAX] or a pathname component is longer than [NAME_MAX].

  [ENOENT]   A component of path does not name an existing file, or the path argument names a nonexistent directory or points to an empty string.
[ENOTDIR]  A component of path is not a directory.

[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.

[EROFS]  The directory entry to be removed resides on a read-only file system.

The rmdir() function may fail if:

[ELOOP]  More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the path argument.

[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].

EXAMPLES

Removing a Directory

The following example shows how to remove a directory named /home/cnd/mod1.

```
#include <unistd.h>
int status;
...
status = rmdir("/home/cnd/mod1");
```

APPLICATION USAGE

None.

RATIONALE

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.

The meaning of deleting pathnameldot is unclear, because the name of the file (directory) in the parent directory to be removed is not clear, particularly in the presence of multiple links to a directory.

The POSIX.1-1990 standard was silent with regard to the behavior of rmdir() when there are multiple hard links to the directory being removed. The requirement to set errno to [EEXIST] or [ENOTEMPTY] clarifies the behavior in this case.

If the process’ current working directory is being removed, that should be an allowed error.

Virtually all existing implementations detect [ENOTEMPTY] or the case of dot-dot. The text in Section 2.3 (on page 21) about returning any one of the possible errors permits that behavior to continue. The [ELOOP] error may be returned if more than {SYMLOOP_MAX} symbolic links are encountered during resolution of the path argument.
FUTURE DIRECTIONS

None.

SEE ALSO
Section 2.3 (on page 21), mkdir(), remove(), unlink(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 6
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

• The DESCRIPTION is updated to indicate the results of naming a symbolic link in path.
• The [EIO] mandatory error condition is added.
• The [ELOOP] mandatory error condition is added.
• A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

• The [ELOOP] optional error condition is added.
NAME
round, roundf, roundl — round to the nearest integer value in a floating-point format

SYNOPSIS
#include <math.h>

double round(double x);
float roundf(float x);
long double roundl(long double x);

DESCRIPTION
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

These functions shall round their argument to the nearest integer value in floating-point format,
rounding halfway cases away from zero, regardless of the current rounding direction.

An application wishing to check for error situations should set errno to zero and call
fetestexcept(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.

RETURN VALUE
Upon successful completion, these functions shall return the rounded integer value.

If x is NaN, a NaN shall be returned.
If x is ±0 or ±Inf, x shall be returned.
If the correct value would cause overflow, a range error shall occur and round(), roundf(), and
roundl() shall return the value of the macro ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL
(with the same sign as x), respectively.

ERRORS
These functions may fail if:

Range Error The result overflows.
If the integer expression (math_errno & MATH_ERRNO) is non-zero,
then errno shall be set to [ERANGE]. If the integer expression
(math_errno & MATH_IEREXCEPT) is non-zero, then the overflow
floating-point exception shall be raised.

EXAMPLES
None.

APPLICATION USAGE
On error, the expressions (math_errno & MATH_ERRNO) and (math_errno &
MATH_IEREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO

feclearexcept(), fetestexcept(), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY
NAME
scalb — load exponent of a radix-independent floating-point number

SYNOPSIS
#include <math.h>

double scalb(double x, double n);

DESCRIPTION
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>.

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.

RETURN VALUE
Upon successful completion, the scalb() function shall return \( x \cdot r^n \).

If \( x \) or \( n \) is NaN, a NaN shall be returned.

If \( n \) is zero, \( x \) shall be returned.

If \( x \) is ±Inf and \( n \) is not –Inf, \( x \) shall be returned.

If \( x \) is ±0 and \( n \) is not +Inf, \( x \) shall be returned.

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.

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.

If the result would cause an overflow, a range error shall occur and ±HUGE_VAL (according to the sign of \( x \)) shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 shall be returned.

ERRORS
The scalb() function shall fail if:

Domain Error
If \( x \) is zero and \( n \) is +Inf, or \( x \) is Inf and \( n \) is –Inf.

If the integer expression (math_errno & MATH_ERRNO) is non-zero, then errno shall be set to [EDOM]. If the integer expression (math_errno & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

Range Error
The result would overflow.

If the integer expression (math_errno & MATH_ERRNO) is non-zero, then errno shall be set to [ERANGE]. If the integer expression (math_errno & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.
The `scalb()` function may fail if:

Range Error  The result underflows.

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.

**EXAMPLES**

None.

**APPLICATION USAGE**

Applications should use either `scalbln()`, `scalblnf()`, or `scalblnl()` in preference to this function.

IEEE Std 1003.1-2001 only defines the behavior for the `scalb()` function when the `n` argument is an integer, a NaN, or Inf. The behavior of other values for the `n` argument is unspecified.

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.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`feclearexcept()`, `fetestexcept()`, `ilogb()`, `ldexp()`, `logb()`, `scalbln()`, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, `<float.h>`, `<math.h>`

**CHANGE HISTORY**

First released in Issue 4, Version 2.

**Issue 5**

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

**Issue 6**

This function is marked obsolescent.

Although this function is not part of the ISO/IEC 9899:1999 standard, the RETURN VALUE and ERRORS sections are updated to align with the error handling in the ISO/IEC 9899:1999 standard.
scalbln, scalblnf, scalblnl, scalbn, scalbnf, scalbnl — compute exponent using FLT_RADIX

#include <math.h>

double scalbln(double x, long n);
float scalblnf(float x, long n);
long double scalblnl(long double x, long n);
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);

These functions shall compute \(x \times \text{FLT\_RADIX}^n\) efficiently, not normally by computing \(\text{FLT\_RADIX}^n\) explicitly.

An application wishing to check for error situations should set \textit{errno} to zero and call \textit{feclearexcept}(FE_ALL_EXCEPT) before calling these functions. On return, if \textit{errno} is non-zero or \textit{fetestexcept}(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

Upon successful completion, these functions shall return \(x \times \text{FLT\_RADIX}^n\).

If the result would cause overflow, a range error shall occur and these functions shall return \pm\text{HUGE\_VAL}, \pm\text{HUGE\_VALF}, and \pm\text{HUGE\_VALL} (according to the sign of \(x\)) as appropriate for the return type of the function.

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.

If \(x\) is NaN, a NaN shall be returned.

If \(x\) is \pm0 or \pm\text{Inf}, \(x\) shall be returned.

If \(n\) is 0, \(x\) shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

These functions shall fail if:

Range Error The result overflows.

If the integer expression (\textit{math\_errhandling} \& \text{MATH\_ERRNO}) is non-zero, then \textit{errno} shall be set to [ERANGE]. If the integer expression (\textit{math\_errhandling} \& \text{MATH\_ERREXCEPT}) is non-zero, then the floating-point exception shall be raised.

These functions may fail if:

Range Error The result underflows.

If the integer expression (\textit{math\_errhandling} \& \text{MATH\_ERRNO}) is non-zero, then \textit{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.

**EXAMPLES**

None.

**APPLICATION USAGE**

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.

**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.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

feclearexcept(), fetestexcept(), scalb(), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

**CHANGE HISTORY**

NAME
scanf — convert formatted input

SYNOPSIS
#include <stdio.h>
int scanf(const char *restrict format, ... );

DESCRIPTION
Refer to fscanf().
NAME
sched_get_priority_max, sched_get_priority_min — get priority limits (REALTIME)

SYNOPSIS

```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

DESCRIPTION
The sched_get_priority_max() and sched_get_priority_min() functions shall return the appropriate maximum or minimum, respectively, for the scheduling policy specified by policy.

The value of policy shall be one of the scheduling policy values defined in <sched.h>.

RETURN VALUE
If successful, the sched_get_priority_max() and sched_get_priority_min() functions shall return the appropriate maximum or minimum values, respectively. If unsuccessful, they shall return a value of −1 and set errno to indicate the error.

ERRORS
The sched_get_priority_max() and sched_get_priority_min() functions shall fail if:

- [EINVAL] The value of the policy parameter does not represent a defined scheduling policy.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
sched_getparam(), sched_setparam(), sched_setscheduler(), sched_rr_get_interval(), sched_setscheduler(), the Base Definitions volume of IEEE Std 1003.1-2001, <sched.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6
These functions are marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

The [ESRCH] error condition has been removed since these functions do not take a pid argument.
NAME
sched_getparam — get scheduling parameters (REALTIME)

SYNOPSIS
#include <sched.h>

int sched_getparam(pid_t pid, struct sched_param *param);

DESCRIPTION
The sched_getparam() function shall return the scheduling parameters of a process specified by
pid in the sched_param structure pointed to by param.

If a process specified by pid exists, and if the calling process has permission, the scheduling
parameters for the process whose process ID is equal to pid shall be returned.

If pid is zero, the scheduling parameters for the calling process shall be returned. The behavior of
the sched_getparam() function is unspecified if the value of pid is negative.

RETURN VALUE
Upon successful completion, the sched_getparam() function shall return zero. If the call to
sched_getparam() is unsuccessful, the function shall return a value of -1 and set errno to indicate
the error.

ERRORS
The sched_getparam() function shall fail if:

[EPERM] The requesting process does not have permission to obtain the scheduling
parameters of the specified process.

[ESRCH] No process can be found corresponding to that specified by pid.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
sched_getscheduler(), sched_setparam(), sched_setscheduler(), the Base Definitions volume of
IEEE Std 1003.1-2001, <sched.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The sched_getparam() function is marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an
implementation does not support the Process Scheduling option.
NAME
sched_getscheduler — get scheduling policy (REALTIME)

SYNOPSIS
#include <sched.h>

int sched_getscheduler(pid_t pid);

DESCRIPTION
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.

The values that can be returned by sched_getscheduler() are defined in the <sched.h> header.

If a process specified by pid exists, and if the calling process has permission, the scheduling policy shall be returned for the process whose process ID is equal to pid.

If pid is zero, the scheduling policy shall be returned for the calling process.

RETURN VALUE
Upon successful completion, the sched_getscheduler() function shall return the scheduling policy of the specified process. If unsuccessful, the function shall return −1 and set errno to indicate the error.

ERRORS
The sched_getscheduler() function shall fail if:

[EPERM] The requesting process does not have permission to determine the scheduling policy of the specified process.

[ESRCH] No process can be found corresponding to that specified by pid.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
sched_getparam(), sched_setparam(), sched_setscheduler(), the Base Definitions volume of IEEE Std 1003.1-2001, <sched.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6
The sched_getscheduler() function is marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.
NAME
sched_rr_get_interval — get execution time limits (REALTIME)

SYNOPSIS
#include <sched.h>

int sched_rr_get_interval(pid_t pid, struct timespec *interval);

DESCRIPTION
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.

RETURN VALUE
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.

ERRORS
The sched_rr_get_interval() function shall fail if:
[ESRCH] No process can be found corresponding to that specified by pid.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
sched_getparam(), sched_get_priority_max(), sched_getscheduler(), sched_setparam(), sched_setscheduler(), the Base Definitions volume of IEEE Std 1003.1-2001, <sched.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6
The sched_rr_get_interval() function is marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.
NAME

sched_setparam — set scheduling parameters (REALTIME)

SYNOPSIS

```c
#include <sched.h>

int sched_setparam(pid_t pid, const struct sched_param *param);
```

DESCRIPTION

The `sched_setparam()` function shall set the scheduling parameters of the process specified by `pid` to the values specified by the `sched_param` structure pointed to by `param`. The value of the `sched_priority` member in the `sched_param` structure shall be any integer within the inclusive priority range for the current scheduling policy of the process specified by `pid`. Higher numerical values for the priority represent higher priorities. If the value of `pid` is negative, the behavior of the `sched_setparam()` function is unspecified.

If a process specified by `pid` exists, and if the calling process has permission, the scheduling parameters shall be set for the process whose process ID is equal to `pid`.

If `pid` is zero, the scheduling parameters shall be set for the calling process.

The conditions under which one process has permission to change the scheduling parameters of another process are implementation-defined.

Implementations may require the requesting process to have the appropriate privilege to set its own scheduling parameters or those of another process.

The target process, whether it is running or not running, shall be moved to the tail of the thread list for its priority.

If the priority of the process specified by the `pid` argument is set higher than that of the lowest priority running process and if the specified process is ready to run, the process specified by the `pid` argument shall preempt a lowest priority running process. Similarly, if the process calling `sched_setparam()` sets its own priority lower than that of one or more other non-empty process lists, then the process that is the head of the highest priority list shall also preempt the calling process. Thus, in either case, the originating process might not receive notification of the completion of the requested priority change until the higher priority process has executed.

If the scheduling policy of the target process is SCHED_SPORADIC, the value specified by the `sched_ss_low_priority` member of the `param` argument shall be any integer within the inclusive priority range for the sporadic server policy. The `sched_ss_repl_period` and `sched_ss_init_budget` members of the `param` argument shall represent the time parameters to be used by the sporadic server scheduling policy for the target process. The `sched_ss_max_repl` member of the `param` argument shall represent the maximum number of replenishments that are allowed to be pending simultaneously for the process scheduled under this scheduling policy.

The specified `sched_ss_repl_period` shall be greater than or equal to the specified `sched_ss_init_budget` for the function to succeed; if it is not, then the function shall fail.

The value of `sched_ss_max_repl` shall be within the inclusive range [1,SS_REPL_MAX] for the function to succeed; if not, the function shall fail.

If the scheduling policy of the target process is either SCHED_FIFO or SCHED_RR, the `sched_ss_low_priority`, `sched_ss_repl_period`, and `sched_ss_init_budget` members of the `param` argument shall have no effect on the scheduling behavior. If the scheduling policy of this process is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the effects of these members are implementation-defined; this case includes the SCHED_OTHER policy.
If the current scheduling policy for the process specified by `pid` is not `SCHED_FIFO`, `SCHED_RR`, or `SCHED_SPORADIC`, the result is implementation-defined; this case includes the `SCHED_OTHER` policy.

The effect of this function on individual threads is dependent on the scheduling contention scope of the threads:

- For threads with system scheduling contention scope, these functions shall have no effect on their scheduling.
- For threads with process scheduling contention scope, the threads’ scheduling parameters shall not be affected. However, the scheduling of these threads with respect to threads in other processes may be dependent on the scheduling parameters of their process, which are governed using these functions.

If an implementation supports a two-level scheduling model in which library threads are multiplexed on top of several kernel-scheduled entities, then the underlying kernel-scheduled entities for the system contention scope threads shall not be affected by these functions.

The underlying kernel-scheduled entities for the process contention scope threads shall have their scheduling parameters changed to the value specified in `param`. Kernel-scheduled entities for use by process contention scope threads that are created after this call completes shall inherit their scheduling policy and associated scheduling parameters from the process.

This function is not atomic with respect to other threads in the process. Threads may continue to execute while this function call is in the process of changing the scheduling policy for the underlying kernel-scheduled entities used by the process contention scope threads.

**RETURN VALUE**

If successful, the `sched_setparam()` function shall return zero.

If the call to `sched_setparam()` is unsuccessful, the priority shall remain unchanged, and the function shall return a value of −1 and set `errno` to indicate the error.

**ERRORS**

The `sched_setparam()` function shall fail if:

- `[EINVAL]` One or more of the requested scheduling parameters is outside the range defined for the scheduling policy of the specified `pid`.
- `[EPERM]` The requesting process does not have permission to set the scheduling parameters for the specified process, or does not have the appropriate privilege to invoke `sched_setparam()`.
- `[ESRCH]` No process can be found corresponding to that specified by `pid`.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.
sched_setparam()

SEE ALSO

sched_getparam(), sched_getscheduler(), sched_setscheduler(), the Base Definitions volume of IEEE Std 1003.1-2001, <sched.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The sched_setparam() function is marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the effect of this function on a thread’s scheduling parameters is added.
- Sections describing two-level scheduling and atomicity of the function are added.

The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #100 is applied.
NAME
sched_setscheduler — set scheduling policy and parameters (REALTIME)

SYNOPSIS

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param);
```

DESCRIPTION

The `sched_setscheduler()` function shall set the scheduling policy and scheduling parameters of the process specified by `pid` to `policy` and the parameters specified in the `sched_param` structure pointed to by `param`, respectively. The value of the `sched_priority` member in the `sched_param` structure shall be any integer within the inclusive priority range for the scheduling policy specified by `policy`. If the value of `pid` is negative, the behavior of the `sched_setscheduler()` function is unspecified.

The possible values for the `policy` parameter are defined in the `<sched.h>` header.

If a process specified by `pid` exists, and if the calling process has permission, the scheduling policy and scheduling parameters shall be set for the process whose process ID is equal to `pid`.

If `pid` is zero, the scheduling policy and scheduling parameters shall be set for the calling process.

The conditions under which one process has the appropriate privilege to change the scheduling parameters of another process are implementation-defined.

Implementations may require that the requesting process have permission to set its own scheduling parameters or those of another process. Additionally, implementation-defined restrictions may apply as to the appropriate privileges required to set a process' own scheduling policy, or another process' scheduling policy, to a particular value.

The `sched_setscheduler()` function shall be considered successful if it succeeds in setting the scheduling policy and scheduling parameters of the process specified by `pid` to the values specified by `policy` and the structure pointed to by `param`, respectively.

If the scheduling policy specified by `policy` is `SCHED_SPORADIC`, the value specified by the `sched_ss_low_priority` member of the `param` argument shall be any integer within the inclusive priority range for the sporadic server policy. The `sched_ss_repl_period` and `sched_ss_init_budget` members of the `param` argument shall represent the time parameters used by the sporadic server scheduling policy for the target process. The `sched_ss_max_repl` member of the `param` argument shall represent the maximum number of replenishments that are allowed to be pending simultaneously for the process scheduled under this scheduling policy.

The specified `sched_ss_repl_period` shall be greater than or equal to the specified `sched_ss_init_budget` for the function to succeed; if it is not, then the function shall fail.

The value of `sched_ss_max_repl` shall be within the inclusive range `[1,SS_REPL_MAX]` for the function to succeed; if not, the function shall fail.

If the scheduling policy specified by `policy` is either `SCHED_FIFO` or `SCHED_RR`, the `sched_ss_low_priority`, `sched_ss_repl_period`, and `sched_ss_init_budget` members of the `param` argument shall have no effect on the scheduling behavior.

The effect of this function on individual threads is dependent on the scheduling contention scope of the threads:
For threads with system scheduling contention scope, these functions shall have no effect on their scheduling.

For threads with process scheduling contention scope, the threads' scheduling policy and associated parameters shall not be affected. However, the scheduling of these threads with respect to threads in other processes may be dependent on the scheduling parameters of their process, which are governed using these functions.

If an implementation supports a two-level scheduling model in which library threads are multiplexed on top of several kernel-scheduled entities, then the underlying kernel-scheduled entities for the system contention scope threads shall not be affected by these functions.

The underlying kernel-scheduled entities for the process contention scope threads shall have their scheduling policy and associated scheduling parameters changed to the values specified in policy and param, respectively. Kernel-scheduled entities for use by process contention scope threads that are created after this call completes shall inherit their scheduling policy and associated scheduling parameters from the process.

This function is not atomic with respect to other threads in the process. Threads may continue to execute while this function call is in the process of changing the scheduling policy and associated scheduling parameters for the underlying kernel-scheduled entities used by the process contention scope threads.

Upon successful completion, the function shall return the former scheduling policy of the specified process. If the sched_setscheduler() function fails to complete successfully, the policy and scheduling parameters shall remain unchanged, and the function shall return a value of −1 and set errno to indicate the error.

The sched_setscheduler() function shall fail if:

- EINVAL The value of the policy parameter is invalid, or one or more of the parameters contained in param is outside the valid range for the specified scheduling policy.
- EPERM The requesting process does not have permission to set either or both of the scheduling parameters or the scheduling policy of the specified process.
- ESRCH No process can be found corresponding to that specified by pid.

None.

None.

None.

None.

None.

sched_getparam(), sched_setscheduler(), sched_setparam(), the Base Definitions volume of IEEE Std 1003.1-2001, <sched.h>
CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The `sched_setscheduler()` function is marked as part of the Process Scheduling option.

The `[ENOSYS]` error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the effect of this function on a thread’s scheduling parameters is added.
- Sections describing two-level scheduling and atomicity of the function are added.

The `SCHED_SPORADIC` scheduling policy is added for alignment with IEEE Std 1003.1d-1999.
NAME
sched_yield — yield the processor

SYNOPSIS
#include <sched.h>

int sched_yield(void);

DESCRIPTION
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.

RETURN VALUE
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.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, < sched.h >

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the
POSIX Threads Extension.

Issue 6
The sched_yield() function is now marked as part of the Process Scheduling and Threads options.
NAME
seed48 — seed a uniformly distributed pseudo-random non-negative long integer generator

SYNOPSIS
XSI
#include <stdlib.h>

unsigned short *seed48(unsigned short seed16v[3]);

DESCRIPTION
Refer to drand48().
NAME
seekdir — set the position of a directory stream

SYNOPSIS
XSI
#include <dirent.h>
void seekdir(DIR *dirp, long loc);

DESCRIPTION
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.

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.

RETURN VALUE
The seekdir() function shall not return a value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
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.

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.

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.

FUTURE DIRECTIONS
None.

SEE ALSO
opendir(), readdir(), telldir(), the Base Definitions volume of IEEE Std 1003.1-2001, <dirent.h>,
<stdlib.h>, <sys/types.h>

CHANGE HISTORY
First released in Issue 2.
In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.
NAME
select — synchronous I/O multiplexing

SYNOPSIS
#include <sys/time.h>

int select(int nfds, fd_set *restrict readfds,
 fd_set *restrict writefds, fd_set *restrict errorfds,
 struct timeval *restrict timeout);

DESCRIPTION
Refer to pselect().
NAME
sem_close — close a named semaphore (REALTIME)

SYNOPSIS
#include <semaphore.h>

int sem_close(sem_t *sem);

DESCRIPTION
The sem_close() function shall indicate that the calling process is finished using the named
semaphore indicated by sem. The effects of calling sem_close() for an unnamed semaphore (one
created by sem_init()) are undefined. The sem_close() function shall deallocate (that is, make
available for reuse by a subseqent sem_open() by this process) any system resources allocated
by the system for use by this process for this semaphore. The effect of subsequent use of the
semaphore indicated by sem by this process is undefined. If the semaphore has not been
removed with a successful call to sem_unlink(), then sem_close() has no effect on the state of the
semaphore. If the sem_unlink() function has been successfully invoked for name after the most
recent call to sem_open() with O_CREAT for this semaphore, then when all processes that have
opened the semaphore close it, the semaphore is no longer accessible.

RETURN VALUE
Upon successful completion, a value of zero shall be returned. Otherwise, a value of −1 shall be
returned and errno set to indicate the error.

ERRORS
The sem_close() function shall fail if:

EINVAL The sem argument is not a valid semaphore descriptor.

APPLICATION USAGE
The sem_close() function is part of the Semaphores option and need not be available on all
implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
semctl(), semget(), semop(), sem_init(), sem_open(), sem_unlink(), the Base Definitions volume of
IEEE Std 1003.1-2001, <semaphore.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6
The sem_close() function is marked as part of the Semaphores option.
The [ENOSYS] error condition has been removed as stubs need not be provided if an
implementation does not support the Semaphores option.
NAME
sem_destroy — destroy an unnamed semaphore (REALTIME)

SYNOPSIS
#include <semaphore.h>

int sem_destroy(sem_t *sem);

DESCRIPTION
The sem_destroy() function shall destroy the unnamed semaphore indicated by sem. Only a
semaphore that was created using sem_init() may be destroyed using sem_destroy(); the effect of
calling sem_destroy() with a named semaphore is undefined. The effect of subsequent use of the
semaphore sem is undefined until sem is reinitialized by another call to sem_init().

It is safe to destroy an initialized semaphore upon which no threads are currently blocked. The
effect of destroying a semaphore upon which other threads are currently blocked is undefined.

RETURN VALUE
Upon successful completion, a value of zero shall be returned. Otherwise, a value of −1 shall be
returned and errno set to indicate the error.

ERRORS
The sem_destroy() function shall fail if:

[EINVVAL] The sem argument is not a valid semaphore.

The sem_destroy() function may fail if:

[EBUSY] There are currently processes blocked on the semaphore.

EXAMPLES
None.

APPLICATION USAGE
The sem_destroy() function is part of the Semaphores option and need not be available on all
implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
semctl(), semget(), semop(), sem_init(), sem_open(), the Base Definitions volume of
IEEE Std 1003.1-2001, <semaphore.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6
The sem_destroy() function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an
implementation does not support the Semaphores option.
NAME
sem_getvalue — get the value of a semaphore (REALTIME)

SYNOPSIS

```c
#include <semaphore.h>

int sem_getvalue(sem_t *restrict sem, int *restrict sval);
```

DESCRIPTION

The `sem_getvalue()` function shall update the location referenced by the `sval` argument to have the value of the semaphore referenced by `sem` without affecting the state of the semaphore. The updated value represents an actual semaphore value that occurred at some unspecified time during the call, but it need not be the actual value of the semaphore when it is returned to the calling process.

If `sem` is locked, then the value returned by `sem_getvalue()` is either zero or a negative number whose absolute value represents the number of processes waiting for the semaphore at some unspecified time during the call.

RETURN VALUE

Upon successful completion, the `sem_getvalue()` function shall return a value of zero. Otherwise, it shall return a value of −1 and set `errno` to indicate the error.

ERRORS

The `sem_getvalue()` function shall fail if:

- [EINVAL] The `sem` argument does not refer to a valid semaphore.

EXAMPLES

None.

APPLICATION USAGE

The `sem_getvalue()` function is part of the Semaphores option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

- `semctl()`, `semget()`, `semop()`, `sem_post()`, `sem_timedwait()`, `sem_trywait()`, `sem_wait()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<semaphore.h>`

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The `sem_getvalue()` function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

The `sem_timedwait()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

The `restrict` keyword is added to the `sem_getvalue()` prototype for alignment with the ISO/IEC 9899:1999 standard.
sem_init()

NAME
sem_init — initialize an unnamed semaphore (REALTIME)

SYNOPSIS

```c
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);
```

DESCRIPTION
The sem_init() function shall initialize the unnamed semaphore referred to by sem. The value of the initialized semaphore shall be value. Following a successful call to sem_init(), the semaphore may be used in subsequent calls to sem_wait(), sem_trywait(), sem_post(), and sem_destroy().

This semaphore shall remain usable until the semaphore is destroyed.

If the pshared argument has a non-zero value, then the semaphore is shared between processes; in this case, any process that can access the semaphore sem can use sem for performing sem_wait(), sem_trywait(), sem_post(), and sem_destroy() operations.

Only sem itself may be used for performing synchronization. The result of referring to copies of sem in calls to sem_wait(), sem_trywait(), sem_post(), and sem_destroy() is undefined.

If the pshared argument is zero, then the semaphore is shared between threads of the process; any thread in this process can use sem for performing sem_wait(), sem_trywait(), sem_post(), and sem_destroy() operations. The use of the semaphore by threads other than those created in the same process is undefined.

Attempting to initialize an already initialized semaphore results in undefined behavior.

RETURN VALUE
Upon successful completion, the sem_init() function shall initialize the semaphore in sem. Otherwise, it shall return −1 and set errno to indicate the error.

ERRORS
The sem_init() function shall fail if:

- EINVAL: The value argument exceeds {SEM_VALUE_MAX}.
- ENOSPC: A resource required to initialize the semaphore has been exhausted, or the limit on semaphores ({SEM_NSEMS_MAX}) has been reached.
- EPERM: The process lacks the appropriate privileges to initialize the semaphore.

EXAMPLES
None.

APPLICATION USAGE
The sem_init() function is part of the Semaphores option and need not be available on all implementations.

RATIONALE
Although this volume of IEEE Std 1003.1-2001 fails to specify a successful return value, it is likely that a later version may require the implementation to return a value of zero if the call to sem_init() is successful.

FUTURE DIRECTIONS
None.
SEE ALSO
sem_destroy(), sem_post(), sem_timedwait(), sem_trywait(), sem_wait(), the Base Definitions volume of IEEE Std 1003.1-2001, <semaphore.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6
The sem_init() function is marked as part of the Semaphores option.
The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.
The sem_timedwait() function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
NAME
sem_open — initialize and open a named semaphore (REALTIME)

SYNOPSIS
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...);

DESCRIPTION
The sem_open() function shall establish a connection between a named semaphore and a process.
Following a call to sem_open() with semaphore name name, the process may reference the
semaphore associated with name using the address returned from the call. This semaphore may
be used in subsequent calls to sem_wait(), sem_trywait(), sem_post(), and sem_close(). The
semaphore remains usable by this process until the semaphore is closed by a successful call to
sem_close(), _exit(), or one of the exec functions.
The oflag argument controls whether the semaphore is created or merely accessed by the call to
sem_open(). The following flag bits may be set in oflag:

O_CREAT This flag is used to create a semaphore if it does not already exist. If O_CREAT is
set and the semaphore already exists, then O_CREAT has no effect, except as noted
under O_EXCL. Otherwise, sem_open() creates a named semaphore. The O_CREAT
flag requires a third and a fourth argument: mode, which is of type mode_t, and
value, which is of type unsigned. The semaphore is created with an initial value of
value. Valid initial values for semaphores are less than or equal to
{SEM_VALUE_MAX}.

The user ID of the semaphore is set to the effective user ID of the process; the
group ID of the semaphore is set to a system default group ID or to the effective
group ID of the process. The permission bits of the semaphore are 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 specified, the
effect is unspecified.

After the semaphore named name has been created by sem_open() with the
O_CREAT flag, other processes can connect to the semaphore by calling
sem_open() with the same value of name.

O_EXCL If O_EXCL and O_CREAT are set, sem_open() fails if the semaphore name exists.
The check for the existence of the semaphore and the creation of the semaphore if
it does not exist are atomic with respect to other processes executing sem_open() with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the
effect is undefined.

If flags other than O_CREAT and O_EXCL are specified in the oflag parameter, the
effect is unspecified.

The name argument points to a string naming a semaphore object. It is unspecified whether the
name appears in the file system and is visible to 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 sem_open() with the same value of name shall refer to the
same semaphore 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.

If a process makes multiple successful calls to sem_open() with the same value for name, the
same semaphore address shall be returned for each such successful call, provided that there
System Interfaces

sem_open()

have been no calls to sem_unlink() for this semaphore.

References to copies of the semaphore produce undefined results.

RETURN VALUE

Upon successful completion, the sem_open() function shall return the address of the semaphore. Otherwise, it shall return a value of SEM_FAILED and set errno to indicate the error. The symbol SEM_FAILED is defined in the <semaphore.h> header. No successful return from sem_open() shall return the value SEM_FAILED.

ERRORS

If any of the following conditions occur, the sem_open() function shall return SEM_FAILED and set errno to the corresponding value:

- [EACCES] The named semaphore exists and the permissions specified by oflag are denied, or the named semaphore does not exist and permission to create the named semaphore is denied.
- [EEXIST] O_CREAT and O_EXCL are set and the named semaphore already exists.
- [EINTR] The sem_open() operation was interrupted by a signal.
- [EINVAL] The sem_open() operation is not supported for the given name, or O_CREAT was specified in oflag and value was greater than [SEM_VALUE_MAX].
- [ENFILE] Too many semaphore descriptors or file descriptors are currently in use by this process.
- [ENAMETOOLONG] The length of the name argument exceeds [PATH_MAX] or a pathname component is longer than [NAME_MAX].
- [ENFILE] Too many semaphores are currently open in the system.
- [ENOENT] O_CREAT is not set and the named semaphore does not exist.
- [ENOSPC] There is insufficient space for the creation of the new named semaphore.

EXAMPLES

None.

APPLICATION USAGE

The sem_open() function is part of the Semaphores option and need not be available on all implementations.

RATIONALE

Early drafts required an error return value of −1 with the type sem_t * for the sem_open() function, which is not guaranteed to be portable across implementations. The revised text provides the symbolic error code SEM_FAILED to eliminate the type conflict.

FUTURE DIRECTIONS

None.

SEE ALSO

semct1(), semget(), semop(), sem_close(), sem_post(), sem_timedwait(), sem_trywait(), sem_unlink(), sem_wait(), the Base Definitions volume of IEEE Std 1003.1-2001, <semaphore.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.
The `sem_open()` function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

The `sem_timedwait()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
NAME
sem_post — unlock a semaphore (REALTIME)

SYNOPSIS
#include <semaphore.h>
int sem_post(sem_t *sem);

DESCRIPTION
The sem_post() function shall unlock the semaphore referenced by sem by performing a
semaphore unlock operation on that semaphore.

If the semaphore value resulting from this operation is positive, then no threads were blocked
waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

If the value of the semaphore resulting from this operation is zero, then one of the threads
blocked waiting for the semaphore shall be allowed to return successfully from its call to
sem_wait(). If the Process Scheduling option is supported, the thread to be unblocked shall be
chosen in a manner appropriate to the scheduling policies and parameters in effect for the
blocked threads. In the case of the schedulers SCHED_FIFO and SCHED_RR, the highest
priority waiting thread shall be unblocked, and if there is more than one highest priority thread
blocked waiting for the semaphore, then the highest priority thread that has been waiting the
longest shall be unblocked. If the Process Scheduling option is not defined, the choice of a thread
to unblock is unspecified.

If the Process Sporadic Server option is supported, and the scheduling policy is
SCHED_SPORADIC, the semantics are as per SCHED_FIFO above.

The sem_post() function shall be reentrant with respect to signals and may be invoked from a
signal-catching function.

RETURN VALUE
If successful, the sem_post() function shall return zero; otherwise, the function shall return −1
and set errno to indicate the error.

ERRORS
The sem_post() function shall fail if:

EINVAL
The sem argument does not refer to a valid semaphore.

APPLICATION USAGE
The sem_post() function is part of the Semaphores option and need not be available on all
implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
semctl(), semget(), semop(), sem_timedwait(), sem_trywait(), sem_wait(), the Base Definitions
volume of IEEE Std 1003.1-2001, <semaphore.h>
CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6
The `sem_post()` function is marked as part of the Semaphores option.
The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.
The `sem_timedwait()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

SCHED_SPORADIC is added to the list of scheduling policies for which the thread that is to be unblocked is specified for alignment with IEEE Std 1003.1d-1999.
NAME

sem_timedwait — lock a semaphore (ADVANCED REALTIME)

SYNOPSIS

#include <semaphore.h>
#include <time.h>

int sem_timedwait(sem_t *restrict sem,
const struct timespec *restrict abs_timeout);

DESCRIPTION

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.

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.

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.

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.

RETURN VALUE

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.

ERRORS

The sem_timedwait() function shall fail if:

EINVAL     The sem argument does not refer to a valid semaphore.
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.
ETIMEDOUT   The semaphore could not be locked before the specified timeout expired.
EDEADLK     A deadlock condition was detected.
EINTR       A signal interrupted this function.
EXAMPLES
None.

APPLICATION USAGE
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.

The sem_timedwait() function is part of the Semaphores and Timeouts options and need not be provided on all implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
sem_post(), sem_trywait(), sem_wait(), semctl(), semget(), semop(), time(), the Base Definitions volume of IEEE Std 1003.1-2001, <semaphore.h>, <time.h>

CHANGE HISTORY
NAME
sem_trywait, sem_wait — lock a semaphore (REALTIME)

SYNOPSIS
#include <semaphore.h>

int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);

DESCRIPTION
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.

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.

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.

The sem_wait() function is interruptible by the delivery of a signal.

RETURN VALUE
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.

ERRORS
The sem_trywait() and sem_wait() functions shall fail if:
EAGAIN The semaphore was already locked, so it cannot be immediately locked by the
sem_trywait() operation (sem_trywait() only).
EINVAL The sem argument does not refer to a valid semaphore.

The sem_trywait() and sem_wait() functions may fail if:
EDEADLK A deadlock condition was detected.
EINVAL A signal interrupted this function.

EXAMPLES
None.

APPLICATION USAGE
Applications using these functions may be subject to priority inversion, as discussed in the Base

The sem_trywait() and sem_wait() functions are part of the Semaphores option and need not be
provided on all implementations.

RATIONALE
None.
sem_tr的研发

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`semctl()`, `semget()`, `semop()`, `sem_post()`, `sem_timedwait()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<semaphore.h>`

**CHANGE HISTORY**
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

**Issue 6**
The `sem_tr` and `sem_wait()` functions are marked as part of the Semaphores option.
The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.
The `sem_timedwait()` function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.
NAME
sem_unlink — remove a named semaphore (REALTIME)

SYNOPSIS
#include <semaphore.h>

int sem_unlink(const char *name);

DESCRIPTION
The sem_unlink() function shall remove the semaphore named by the string name. If the
semaphore named by name is currently referenced by other processes, then sem_unlink() shall
have no effect on the state of the semaphore. If one or more processes have the semaphore open
when sem_unlink() is called, destruction of the semaphore is postponed until all references to the
semaphore have been destroyed by calls to sem_close(), _exit(), or exec. Calls to sem_open() to
recreate or reconnect to the semaphore refer to a new semaphore after sem_unlink() is called. The
sem_unlink() call shall not block until all references have been destroyed; it shall return
immediately.

RETURN VALUE
Upon successful completion, the sem_unlink() function shall return a value of 0. Otherwise, the
semaphore shall not be changed and the function shall return a value of −1 and set errno to
indicate the error.

ERRORS
The sem_unlink() function shall fail if:

[EACCES] Permission is denied to unlink the named semaphore.

[ENAMETOOLONG] The length of the name argument exceeds [PATH_MAX] or a pathname
component is longer than [NAME_MAX].

[ENOENT] The named semaphore does not exist.

EXAMPLES
None.

APPLICATION USAGE
The sem_unlink() function is part of the Semaphores option and need not be available on all
implementations.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
semctl(), semget(), semop(), sem_close(), sem_open(), the Base Definitions volume of
IEEE Std 1003.1-2001, <semaphore.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6
The sem_unlink() function is marked as part of the Semaphores option.
The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.
NAME
sem_wait — lock a semaphore (REALTIME)

SYNOPSIS

```c
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

DESCRIPTION
Refer to `sem_trywait()`.
NAME
semctl — XSI semaphore control operations

SYNOPSIS
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);

DESCRIPTION
The semctl() function operates on XSI semaphores (see the Base Definitions volume of
IEEE Std 1003.1-2001, Section 4.15, Semaphore). It is unspecified whether this function
interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
page 41).

The semctl() function provides a variety of semaphore control operations as specified by cmd.
The fourth argument is optional and depends upon the operation requested. If required, it is of
type union semun, which the application shall explicitly declare:

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
} arg;

The following semaphore control operations as specified by cmd are executed with respect to the
semaphore specified by semid and semnum. The level of permission required for each operation
is shown with each command; see Section 2.7 (on page 39). The symbolic names for the values
of cmd are defined in the <sys/sem.h> header:

GETVAL Return the value of semval; see <sys/sem.h>. Requires read permission.

SETVAL Set the value of semval to arg.val, where arg is the value of the fourth argument
to semctl(). When this command is successfully executed, the semadj value
 corresponding to the specified semaphore in all processes is cleared. Requires
alter permission; see Section 2.7 (on page 39).

GETPID Return the value of sempid. Requires read permission.

GETNCNT Return the value of semncnt. Requires read permission.

GETZCNT Return the value of semzcnt. Requires read permission.

The following values of cmd operate on each semval in the set of semaphores:

GETALL Return the value of semval for each semaphore in the semaphore set and place
 into the array pointed to by arg.array, where arg is the fourth argument to
semctl(). Requires read permission.

SETALL Set the value of semval for each semaphore in the semaphore set according to
the array pointed to by arg.array, where arg is the fourth argument to semctl().
When this command is successfully executed, the semadj values corresponding
to each specified semaphore in all processes are cleared. Requires alter
permission.

The following values of cmd are also available:

IPC_STAT Place the current value of each member of the semid_ds data structure
associated with semid into the structure pointed to by arg.buf, where arg is the
fourth argument to semctl(). The contents of this structure are defined in
semctl() requires read permission.

Set the value of the following members of the semid_ds data structure associated with semid to the corresponding value found in the structure pointed to by arg.buf, where arg is the fourth argument to semctl():

- sem_perm.uid
- sem_perm.gid
- sem_perm.mode

The mode bits specified in Section 2.7.1 (on page 40) are copied into the corresponding bits of the sem_perm.mode associated with semid. The stored values of any other bits are unspecified.

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 sem_perm.cuid or sem_perm.uid in the semid_ds data structure associated with semid.

Remove the semaphore identifier specified by semid 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 sem_perm.cuid or sem_perm.uid in the semid_ds data structure associated with semid.

If successful, the value returned by semctl() depends on cmd as follows:

- GETVAL: The value of semval.
- GETPID: The value of sempid.
- GETNCNT: The value of semncnt.
- GETZCNT: The value of semzcnt.
- All others: 0.

Otherwise, semctl() shall return −1 and set errno to indicate the error.

The semctl() function shall fail if:

- [EACCES]: Operation permission is denied to the calling process; see Section 2.7 (on page 39).
- [EINVAL]: The value of semid is not a valid semaphore identifier, or the value of semnum is less than 0 or greater than or equal to sem_nsems, or the value of cmd is not a valid command.
- [EPERM]: The argument cmd 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 sem_perm.cuid or sem_perm.uid in the data structure associated with semid.
- [ERANGE]: The argument cmd is equal to SETVAL or SETALL and the value to which semval is to be set is greater than the system-imposed maximum.
EXAMPLES

None.

APPLICATION USAGE

The fourth parameter in the SYNOPSIS section is now specified as "..." in order to avoid a clash with the ISO C standard when referring to the union semun (as defined in Issue 3) and for backwards-compatibility.

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.7 (on page 39), Section 2.8 (on page 41), semget(), semop(), sem_close(), sem_destroy(), sem_getvalue(), sem_init(), sem_open(), sem_post(), sem_unlink(), sem_wait(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/sem.h>

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to the APPLICATION USAGE section.
NAME
semget — get set of XSI semaphores

SYNOPSIS
XSI
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);

DESCRIPTION
The semget() function operates on XSI semaphores (see the Base Definitions volume of
IEEE Std 1003.1-2001, Section 4.15, Semaphore). It is unspecified whether this function
interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on
page 41).

The semget() function shall return the semaphore identifier associated with key.

A semaphore identifier with its associated semid_ds data structure and its associated set of
nsems semaphores (see <sys/sem.h>) is created for key if one of the following is true:

• The argument key is equal to IPC_PRIVATE.
• The argument key does not already have a semaphore identifier associated with it and (semflg
 &IPC_CREAT) is non-zero.

Upon creation, the semid_ds data structure associated with the new semaphore identifier is
initialized as follows:

• In the operation permissions structure sem_perm.cuid, sem_perm.uid, sem_perm.cgid, and
sem_perm.gid shall be set equal to the effective user ID and effective group ID, respectively, of
the calling process.
• The low-order 9 bits of sem_perm.mode shall be set equal to the low-order 9 bits of semflg.
• The variable sem_nsems shall be set equal to the value of nsems.
• The variable sem_otime shall be set equal to 0 and sem_ctime shall be set equal to the current
time.
• The data structure associated with each semaphore in the set shall not be initialized. The
semctl() function with the command SETVAL or SETALL can be used to initialize each
semaphore.

RETURN VALUE
Upon successful completion, semget() shall return a non-negative integer, namely a semaphore
identifier; otherwise, it shall return −1 and set errno to indicate the error.

ERRORS
The semget() function shall fail if:

[EACCES] A semaphore identifier exists for key, but operation permission as specified by
the low-order 9 bits of semflg would not be granted; see Section 2.7 (on page
39).

[EXIST] A semaphore identifier exists for the argument key but ((semflg &IPC_CREAT)
&(&semflg &IPC_EXCL)) is non-zero.

[EINVAL] The value of nsems is either less than or equal to 0 or greater than the system-
imposed limit, or a semaphore identifier exists for the argument key, but the
number of semaphores in the set associated with it is less than nsems and
nsems is not equal to 0.
semget( )

A semaphore identifier does not exist for the argument key and (semflg &IPC_CREAT) is equal to 0.

A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system-wide would be exceeded.

**EXAMPLES**

**Creating a Semaphore Identifier**

The following example gets a unique semaphore key using the ftok() function, then gets a semaphore ID associated with that key using the semget() function (the first call also tests to make sure the semaphore exists). If the semaphore does not exist, the program creates it, as shown by the second call to semget(). In creating the semaphore for the queuing process, the program attempts to create one semaphore with read/write permission for all. It also uses the IPC_EXCL flag, which forces semget() to fail if the semaphore already exists.

After creating the semaphore, the program uses a call to semop() to initialize it to the values in the sbuf array. The number of processes that can execute concurrently without queuing is initially set to 2. The final call to semget() creates a semaphore identifier that can be used later in the program.

```c
#include <sys/types.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <pwd.h>
#include <fcntl.h>
#include <limits.h>
...
key_t semkey;
int semid, pfd, fv;
struct sembuf sbuf;
char *lgn;
char filename[PATH_MAX+1];
struct stat outstat;
struct passwd *pw;
...
/* Get unique key for semaphore. */
if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
    perror("IPC error: ftok"); exit(1);
}
/* Get semaphore ID associated with this key. */
if ((semid = semget(semkey, 0, 0)) == -1) {
    /* Semaphore does not exist - Create. */
    if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
        S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
    {
        /* Initialize the semaphore. */
        sbuf.sem_num = 0;
    }
```

---

**System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.**
... sbuf.sem_op = 2; /* This is the number of runs
without queuing */
sbuf.sem_flg = 0;
if (semop(semid, &sbuf, 1) == -1) {
perror("IPC error: semop"); exit(1);
}
else if (errno == EEXIST) {
    if ((semid = semget(semkey, 0, 0)) == -1) {
perror("IPC error 1: semget"); exit(1);
    }
}
else {
    perror("IPC error 2: semget"); exit(1);
}
...

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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.7 (on page 39), Section 2.8 (on page 41), semctl(), semop(), sem_close(), sem_destroy(),
sem_getvalue(), sem_init(), sem_open(), sem_post(), sem_unlink(), sem_wait(), the Base Definitions
volume of IEEE Std 1003.1-2001, <sys/sem.h>

CHANGE HISTORY
First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5
The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
DIRECTIONS to a new APPLICATION USAGE section.
**NAME**
semop — XSI semaphore operations

**SYNOPSIS**

```c
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
```

**DESCRIPTION**

The `semop()` function operates on XSI semaphores (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.15, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 41).

The `semop()` function shall perform atomically a user-defined array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by the argument `semid`.

The argument `sops` is a pointer to a user-defined array of semaphore operation structures. The implementation shall not modify elements of this array unless the application uses implementation-defined extensions.

The argument `nsops` is the number of such structures in the array.

Each structure, `sembuf`, includes the following members:

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>sem_num</td>
<td>Semaphore number.</td>
</tr>
<tr>
<td>short</td>
<td>sem_op</td>
<td>Semaphore operation.</td>
</tr>
<tr>
<td>short</td>
<td>sem_flg</td>
<td>Operation flags.</td>
</tr>
</tbody>
</table>

Each semaphore operation specified by `sem_op` is performed on the corresponding semaphore specified by `semid` and `sem_num`.

The variable `sem_op` specifies one of three semaphore operations:

1. If `sem_op` is a negative integer and the calling process has alter permission, one of the following shall occur:

   - If `semval` (see `<sys/sem.h>`) is greater than or equal to the absolute value of `sem_op`, the absolute value of `sem_op` is subtracted from `semval`. Also, if `(sem_flg & SEM_UNDO)` is non-zero, the absolute value of `sem_op` shall be added to the calling process' `semadj` value for the specified semaphore.

   - If `semval` is less than the absolute value of `sem_op` and `(sem_flg & IPC_NOWAIT)` is non-zero, `semop()` shall return immediately.

   - If `semval` is less than the absolute value of `sem_op` and `(sem_flg & IPC_NOWAIT)` is 0, `semop()` shall increment the `semncnt` associated with the specified semaphore and suspend execution of the calling thread until one of the following conditions occurs:

     - The value of `semval` becomes greater than or equal to the absolute value of `sem_op`. When this occurs, the value of `semncnt` associated with the specified semaphore shall be decremented, the absolute value of `sem_op` shall be subtracted from `semval` and, if `(sem_flg & SEM_UNDO)` is non-zero, the absolute value of `sem_op` shall be added to the calling process' `semadj` value for the specified semaphore.

     - The `semid` for which the calling thread is awaiting action is removed from the system. When this occurs, `errno` shall be set equal to [EIDRM] and −1 shall be
The calling thread receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore shall be decremented, and the calling thread shall resume execution in the manner prescribed in sigaction().

2. If sem_op is a positive integer and the calling process has alter permission, the value of sem_op shall be added to semval and, if (sem_flg & SEM_UNDO) is non-zero, the value of sem_op shall be subtracted from the calling process' semadj value for the specified semaphore.

3. If sem_op is 0 and the calling process has read permission, one of the following shall occur:
   • If semval is 0, semop() shall return immediately.
   • If semval is non-zero and (sem_flg & IPC_NOWAIT) is non-zero, semop() shall return immediately.
   • If semval is non-zero and (sem_flg & IPC_NOWAIT) is 0, semop() shall increment the semzcnt associated with the specified semaphore and suspend execution of the calling thread until one of the following occurs:
     — The value of semval becomes 0, at which time the value of semzcnt associated with the specified semaphore shall be decremented.
     — The semid for which the calling thread is awaiting action is removed from the system. When this occurs, errno shall be set equal to [EIDRM] and −1 shall be returned.
     — The calling thread receives a signal that is to be caught. When this occurs, the value of semzcnt associated with the specified semaphore shall be decremented, and the calling thread shall resume execution in the manner prescribed in sigaction().

Upon successful completion, the value of sempid for each semaphore specified in the array pointed to by sops shall be set equal to the process ID of the calling process.

RETURN VALUE

Upon successful completion, semop() shall return 0; otherwise, it shall return −1 and set errno to indicate the error.

ERRORS

The semop() function shall fail if:

[E2BIG]  The value of nsops is greater than the system-imposed maximum.
[EACCES]  Operation permission is denied to the calling process; see Section 2.7 (on page 39).
[EAGAIN]  The operation would result in suspension of the calling process but (sem_flg & IPC_NOWAIT) is non-zero.
[EFBIG]  The value of sem_num is less than 0 or greater than or equal to the number of semaphores in the set associated with semid.
[EIDRM]  The semaphore identifier semid is removed from the system.
[EINTR]  The semop() function was interrupted by a signal.
[EINVAL]  The value of semid 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.
semop()

System Interfaces

[ENOSPC] The limit on the number of individual processes requesting a SEM_UNDO would be exceeded.

[ERANGE] An operation would cause a semval to overflow the system-imposed limit, or an operation would cause a semadj value to overflow the system-imposed limit.

EXAMPLES

Setting Values in Semaphores

The following example sets the values of the two semaphores associated with the semid identifier to the values contained in the sb array.

```c
#include <sys/sem.h>
...
int semid;
struct sembuf sb[2];
int nsops = 2;
int result;
/* Adjust value of semaphore in the semaphore array semid. */
sb[0].sem_num = 0;
sb[0].sem_op = -1;
sb[0].sem_flg = SEM_UNDO | IPC_NOWAIT;
sb[1].sem_num = 1;
sb[1].sem_op = 1;
sb[1].sem_flg = 0;
result = semop(semid, sb, nsops);
```

Creating a Semaphore Identifier

The following example gets a unique semaphore key using the ftok() function, then gets a semaphore ID associated with that key using the semget() function (the first call also tests to make sure the semaphore exists). If the semaphore does not exist, the program creates it, as shown by the second call to semget(). In creating the semaphore for the queuing process, the program attempts to create one semaphore with read/write permission for all. It also uses the IPC_EXCL flag, which forces semget() to fail if the semaphore already exists.

After creating the semaphore, the program uses a call to semop() to initialize it to the values in the sbuf array. The number of processes that can execute concurrently without queuing is initially set to 2. The final call to semget() creates a semaphore identifier that can be used later in the program.

The final call to semop() acquires the semaphore and waits until it is free; the SEM_UNDO option releases the semaphore when the process exits, waiting until there are less than two processes running concurrently.

```c
#include <sys/types.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
```
#include <pwd.h>
#include <fcntl.h>
#include <limits.h>

key_t semkey;
int semid, pfds, fv;
struct sembuf sbuf;
char *lgn;
char filename[PATH_MAX+1];
struct stat outstat;
struct passwd *pw;

/* Get unique key for semaphore. */
if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
    perror("IPC error: ftok"); exit(1);
}

/* Get semaphore ID associated with this key. */
if ((semid = semget(semkey, 0, 0)) == -1) {
    /* Semaphore does not exist - Create. */
    if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
        S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1) {
        /* Initialize the semaphore. */
        sbuf.sem_num = 0;
        sbuf.sem_op = 2; /* This is the number of runs without queuing. */
        sbuf.sem_flg = 0;
        if (semop(semid, &sbuf, 1) == -1) {
            perror("IPC error: semop"); exit(1);
        }
    }
    else if (errno == EEXIST) {
        if ((semid = semget(semkey, 0, 0)) == -1) {
            perror("IPC error 1: semget"); exit(1);
        }
    }
    else {
        perror("IPC error 2: semget"); exit(1);
    }
}

... 

sbuf.sem_num = 0;
sbuf.sem_op = -1;
sbuf.sem_flg = SEM_UNDO;
if (semop(semid, &sbuf, 1) == -1) {
    perror("IPC Error: semop"); exit(1);
}

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.
semop()

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.7 (on page 39), Section 2.8 (on page 41), exec, exit(), fork(), semct1(), semget(), sem_close(), sem_destroy(), sem_getvalue(), sem_init(), sem_open(), sem_post(), sem_unlink(), sem_wait(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/ipc.h>, <sys/sem.h>, <sys/types.h>

CHANGE HISTORY
First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5
The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.
NAME
send — send a message on a socket

SYNOPSIS
#include <sys/socket.h>
ssize_t send(int socket, const void *buffer, size_t length, int flags);

DESCRIPTION
The send() function shall initiate transmission of a message from the specified socket to its peer. The send() function shall send a message only when the socket is connected (including when the peer of a connectionless socket has been set via connect()).

The send() function takes the following arguments:
socket Specifies the socket file descriptor.
buffer Points to the buffer containing the message to send.
length Specifies the length of the message in bytes.
flags Specifies the type of message transmission. Values of this argument are formed by logically OR’ing zero or more of the following flags:

- MSG_EOR Terminates a record (if supported by the protocol).
- 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.

The length of the message to be sent is specified by the length argument. If the message is too long to pass through the underlying protocol, send() shall fail and no data shall be transmitted.

Successful completion of a call to send() does not guarantee delivery of the message. A return value of −1 indicates only locally-detected errors.

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, send() 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, send() shall fail. The select() and poll() functions can be used to determine when it is possible to send more data.

The socket in use may require the process to have appropriate privileges to use the send() function.

RETURN VALUE
Upon successful completion, send() shall return the number of bytes sent. Otherwise, −1 shall be returned and errno set to indicate the error.

ERRORS
The send() function shall fail if:

- [EAGAIN] or [EWOULDBLOCK] The socket’s file descriptor is marked O_NONBLOCK and the requested operation would block.
- [EBADF] The socket argument is not a valid file descriptor.
- [ECONNRESET] A connection was forcibly closed by a peer.
- [EDEADDRREQ] The socket is not connection-mode and no peer address is set.
send()  

System Interfaces

39618 [EINTR] A signal interrupted send() before any data was transmitted.
39619 [EMSGSIZE] The message is too large to be sent all at once, as the socket requires.
39620 [ENOTCONN] The socket is not connected or otherwise has not had the peer pre-specified.
39621 [ENOTSOCK] The socket argument does not refer to a socket.
39622 [EOPNOTSUPP] The socket argument is associated with a socket that does not support one or more of the values set in flags.
39623 [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.
39624 The send() function may fail if:
39625 [EACCES] The calling process does not have the appropriate privileges.
39626 [EIO] An I/O error occurred while reading from or writing to the file system.
39627 [ENETDOWN] The local network interface used to reach the destination is down.
39628 [ENETUNREACH] No route to the network is present.
39629 [ENOBUFS] Insufficient resources were available in the system to perform the operation.
39630 EXAMPLES None.
39631 APPLICATION USAGE The send() function is equivalent to sendto() with a null pointer dest_len argument, and to write() if no flags are used.
39632 RATIONALE None.
39633 FUTURE DIRECTIONS None.
39634 SEE ALSO connect(), getsockopt(), poll(), recv(), recvfrom(), recvmsg(), select(), sendmsg(), sendto(), setsockopt(), shutdown(), socket(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>
39635 CHANGE HISTORY First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
sendmsg() — send a message on a socket using a message structure

**SYNOPSIS**
```
#include <sys/socket.h>
ssize_t sendmsg(int socket, const struct msghdr *message, int flags);
```

**DESCRIPTION**
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.

The `sendmsg()` function takes the following arguments:
- `socket` Specifies the socket file descriptor.
- `message` 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 `msg_flags` member is ignored.
- `flags` Specifies the type of message transmission. The application may specify 0 or the following flag:
  - `MSG_EOR` Terminates a record (if supported by the protocol).
  - `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.

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.

Successful completion of a call to `sendmsg()` does not guarantee delivery of the message. A return value of −1 indicates only locally-detected errors.

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.

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.

The socket in use may require the process to have appropriate privileges to use the `sendmsg()` function.

**RETURN VALUE**
Upon successful completion, `sendmsg()` shall return the number of bytes sent. Otherwise, −1 shall be returned and `errno` set to indicate the error.

**ERRORS**
The `sendmsg()` function shall fail if:
- [EAGAIN] or [EWOULDBLOCK]

The socket’s file descriptor is marked O_NONBLOCK and the requested
operation would block.

[EAFNOSUPPORT] Addresses in the specified address family cannot be used with this socket.

[EBADF] The socket argument is not a valid file descriptor.

[ECONNRESET] A connection was forcibly closed by a peer.

[EINTR] A signal interrupted sendmsg() before any data was transmitted.

[EINVAL] The sum of the iov_len values overflows an ssize_t.

[EMSgsizE] The message is too large to be sent all at once (as the socket requires), or the msg_iovlen member of the msghdr structure pointed to by message is less than or equal to 0 or is greater than {IOV_MAX}.

[ENOTCONN] The socket is connection-mode but is not connected.

[ENOTSOCK] The socket argument does not refer to a socket.

[EOPNOTSUPP] The socket argument is associated with a socket that does not support one or more of the values set in flags.

[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.

If the address family of the socket is AF_UNIX, then sendmsg() shall fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

[ELOOP] A loop exists in symbolic links encountered during resolution of the pathname in the socket address.


[ENOENT] A component of the pathname does not name an existing file or the path name is an empty string.

[ENOTDIR] A component of the path prefix of the pathname in the socket address is not a directory.

The sendmsg() function may fail if:

[EACCES] Search permission is denied for a component of the path prefix; or write access to the named socket is denied.

[EDESTADDRREQ] The socket is not connection-mode and does not have its peer address set, and no destination address was specified.

[EHOSTUNREACH] The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).

[EIO] An I/O error occurred while reading from or writing to the file system.

[EISCONN] A destination address was specified and the socket is already connected.

[ENETDOWN] The local network interface used to reach the destination is down.
sendmsg()

39733  [ENETUNREACH]  No route to the network is present.
39735  [ENOBUFFS]    Insufficient resources were available in the system to perform the operation.
39736  [ENOMEM]      Insufficient memory was available to fulfill the request.
39737  If the address family of the socket is AF_UNIX, then sendmsg() may fail if:
39738  [ELOOP]       More than [SYMLOOP_MAX] symbolic links were encountered during resolution of the pathname in the socket address.
39740  [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds [PATH_MAX].

EXAMPLES

Done.

APPLICATION USAGE

The select() and poll() functions can be used to determine when it is possible to send more data.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getsockopt(), poll(), recv(), recvfrom(), recvmsg(), select(), send(), sendto(), setsockopt(), shutdown(), socket(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.
NAME
sendto — send a message on a socket

SYNOPSIS
#include <sys/socket.h>
ssize_t sendto(int socket, const void *message, size_t length,
    int flags, const struct sockaddr *dest_addr,
    socklen_t dest_len);

DESCRIPTION
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.
The sendto() function takes the following arguments:
socket Specifies the socket file descriptor.
message Points to a buffer containing the message to be sent.
length Specifies the size of the message in bytes.
flags Specifies the type of message transmission. Values of this argument are formed by logically OR’ing zero or more of the following flags:
MSG_EOR Terminates a record (if supported by the protocol).
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.
dest_addr Points to a sockaddr structure containing the destination address. The length
and format of the address depend on the address family of the socket.
dest_len Specifies the length of the sockaddr structure pointed to by the dest_addr
argument.

If the socket protocol supports broadcast and the specified address is a broadcast address for the
socket protocol, sendto() shall fail if the SO_BROADCAST option is not set for the socket.
The dest_addr argument specifies the address of the target. The length argument specifies the
length of the message.
Successful completion of a call to sendto() does not guarantee delivery of the message. A return
value of −1 indicates only locally-detected errors.
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, sendto() 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, sendto() shall fail.
The socket in use may require the process to have appropriate privileges to use the sendto() function.

RETURN VALUE
Upon successful completion, sendto() shall return the number of bytes sent. Otherwise, −1 shall
be returned and errno set to indicate the error.
The `sendto()` function shall fail if:

- **[EAFNOSUPPORT]** Addresses in the specified address family cannot be used with this socket.
- **[EAGAIN]** or **[EWOULDBLOCK]** The socket’s file descriptor is marked O_NONBLOCK and the requested operation would block.
- **[EBADF]** The socket argument is not a valid file descriptor.
- **[ECONNRESET]** A connection was forcibly closed by a peer.
- **[EINVAL]** The `dest_len` argument is not a valid length for the address family.
- **[EIO]** An I/O error occurred while reading from or writing to the file system.
- **[ELOOP]** A loop exists in symbolic links encountered during resolution of the pathname in the socket address.
- **[ENAMETOOLONG]** A component of a pathname exceeded `NAME_MAX` characters, or an entire pathname exceeded `PATH_MAX` characters.
- **[ENOENT]** A component of the pathname does not name an existing file or the pathname is an empty string.
- **[ENOTDIR]** A component of the path prefix of the pathname in the socket address is not a directory.
- **[EACCES]** Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
- **[EDESTADDRREQ]** The socket is not connection-mode and does not have its peer address set, and no destination address was specified.
- **[EHOSTUNREACH]** The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
- **[EINVAL]** The `dest_len` argument is not a valid length for the address family.
- **[EIO]** An I/O error occurred while reading from or writing to the file system.
sendto()

[EISCONN] A destination address was specified and the socket is already connected. This error may or may not be returned for connection mode sockets.

[ENETDOWN] The local network interface used to reach the destination is down.

[ENETUNREACH] No route to the network is present.

[ENOBUFFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

If the address family of the socket is AF_UNIX, then sendto() may fail if:

[ELOOP] More than [SYMLOOP_MAX] symbolic links were encountered during resolution of the pathname in the socket address.

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds [PATH_MAX].

EXAMPLES
None.

APPLICATION USAGE
The select() and poll() functions can be used to determine when it is possible to send more data.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getsockopt(), poll(), recv(), recvfrom(), recvmsg(), select(), send(), sendmsg(), setsockopt(), shutdown(), socket(), the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<sys/socket.h>}

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.
NAME
setbuf — assign buffering to a stream

SYNOPSIS
#include <stdio.h>
void setbuf(FILE *restrict stream, char *restrict buf);

DESCRIPTION
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

Except that it returns no value, the function call:
setbuf(stream, buf)
shall be equivalent to:
setvbuf(stream, buf, _IOFBF, BUFSIZ)
if buf is not a null pointer, or to:
setvbuf(stream, buf, _IONBF, BUFSIZ)
if buf is a null pointer.

RETURN VALUE
The setbuf() function shall not return a value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
A common source of error is allocating buffer space as an “automatic” variable in a code block,
and then failing to close the stream in the same block.
With setbuf(), allocating a buffer of BUFSIZ bytes does not necessarily imply that all of BUFSIZ
bytes are used for the buffer area.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fopen(), setvbuf(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The prototype for setbuf() is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
setcontext — set current user context

SYNOPSIS
XSI
#include <ucontext.h>

int setcontext(const ucontext_t *ucp);

DESCRIPTION
Refer to getcontext().
NAME
setegid — set the effective group ID

SYNOPSIS
#include <unistd.h>

int setegid(gid_t gid);

DESCRIPTION
If gid is equal to the real group ID or the saved set-group-ID, or if the process has appropriate
privileges, setegid() shall set the effective group ID of the calling process to gid; the real group
ID, saved set-group-ID, and any supplementary group IDs shall remain unchanged.
The setegid() function shall not affect the supplementary group list in any way.

RETURN VALUE
Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned and errno set to
indicate the error.

ERRORS
The setegid() function shall fail if:

[EINVAL] The value of the gid argument is invalid and is not supported by the
implementation.

[EPERM] The process does not have appropriate privileges and gid does not match the
real group ID or the saved set-group-ID.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
Refer to the RATIONALE section in setuid().

FUTURE DIRECTIONS
None.

SEE ALSO
exec, getegid(), geteuid(), getgid(), getuid(), seteuid(), setgid(), setregid(), setreuid(), setuid(), the
Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 6. Derived from the IEEE P1003.1a draft standard.
NAME
    setenv — add or change environment variable

SYNOPSIS
    #include <stdlib.h>
    int setenv(const char *envname, const char *envval, int overwrite);

DESCRIPTION
    The setenv() function shall update or add a variable in the environment of the calling process. The
    envname argument points to a string containing the name of an environment variable to be
    added or altered. The environment variable shall be set to the value to which envval points. The
    function shall fail if envname points to a string which contains an ‘=’ character. If the environment
    variable named by envname already exists and the value of overwrite is non-zero, the function shall
    return success and the environment shall be updated. If the environment variable named by envname already exists and the value of overwrite is zero, the function shall return success and the environment shall remain unchanged.

    If the application modifies environ or the pointers to which it points, the behavior of setenv() is undefined. The setenv() function shall update the list of pointers to which environ points.

    The strings described by envname and envval are copied by this function.

    The setenv() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE
    Upon successful completion, zero shall be returned. Otherwise, −1 shall be returned, errno set to
    indicate the error, and the environment shall be unchanged.

ERRORS
    The setenv() function shall fail if:

    [EINVAL] The name argument is a null pointer, points to an empty string, or points to a
    string containing an ‘=’ character.

    [ENOMEM] Insufficient memory was available to add a variable or its value to the
    environment.

EXAMPLES
    None.

APPLICATION USAGE
    None.

RATIONALE
    Unanticipated results may occur if setenv() changes the external variable environ. In particular,
    if the optional envp argument to main() is present, it is not changed, and thus may point to an
    obsolete copy of the environment (as may any other copy of environ). However, other than the
    aforementioned restriction, the developers of IEEE Std 1003.1-2001 intended that the traditional
    method of walking through the environment by way of the environ pointer must be supported.

    It was decided that setenv() should be required by this revision because it addresses a piece of
    missing functionality, and does not impose a significant burden on the implementor.

    There was considerable debate as to whether the System V putenv() function or the BSD setenv() function should be required as a mandatory function. The setenv() function was chosen because it permitted the implementation of the unsetenv() function to delete environmental variables, without specifying an additional interface. The putenv() function is available as an XSI
The standard developers considered requiring that `setenv()` 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 `exec`, which returns an indication of this error condition.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`getenv()`, `unsetenv()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`, `<sys/types.h>`, `<unistd.h>`

**CHANGE HISTORY**

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.
NAME
seteuid — set effective user ID

SYNOPSIS
#include <unistd.h>
int seteuid(uid_t uid);

DESCRIPTION
If uid is equal to the real user ID or the saved set-user-ID, or if the process has appropriate
privileges, seteuid() shall set the effective user ID of the calling process to uid; the real user ID
and saved set-user-ID shall remain unchanged.
The seteuid() function shall not affect the supplementary group list in any way.

RETURN VALUE
Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned and errno set to
indicate the error.

ERRORS
The seteuid() function shall fail if:

[EINVAL] The value of the uid argument is invalid and is not supported by the
implementation.

[EPERM] The process does not have appropriate privileges and uid does not match the
real group ID or the saved set-group-ID.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
Refer to the RATIONALE section in setuid().

FUTURE DIRECTIONS
None.

SEE ALSO
exe, getegid(), geteuid(), getgid(), getuid(), setegid(), setgid(), setreuid(), setreuid(), setuid(), the
Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 6. Derived from the IEEE P1003.1a draft standard.
NAME
setgid — set-group-ID

SYNOPSIS
#include <unistd.h>
int setgid(gid_t gid);

DESCRIPTION
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.

If the process does not have appropriate privileges, but gid is equal to the real group ID or the saved set-group-ID, setgid() shall set the effective group ID to gid; the real group ID and saved set-group-ID shall remain unchanged.

The setgid() function shall not affect the supplementary group list in any way.

Any supplementary group IDs of the calling process shall remain unchanged.

RETURN VALUE
Upon successful completion, 0 is returned. Otherwise, −1 shall be returned and errno set to indicate the error.

ERRORS
The setgid() function shall fail if:

[EINVAL] The value of the gid argument is invalid and is not supported by the implementation.

[EPERM] The process does not have appropriate privileges and gid does not match the real group ID or the saved set-group-ID.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
Refer to the RATIONALE section in setuid().

FUTURE DIRECTIONS
None.

SEE ALSO
exec, getegid(), geteuid(), getgid(), getuid(), setegid(), seteuid(), setgid(), setreuid(), setregid(), setreuid(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

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.
• Functionality associated with _POSIX_SAVED_IDS is now mandated. This is a FIPS requirement.

The following changes were made to align with the IEEE P1003.1a draft standard:

• The effects of `setgid()` in processes without appropriate privileges are changed.

• A requirement that the supplementary group list is not affected is added.
NAME
setgrent — reset the group database to the first entry

SYNOPSIS
#include <grp.h>
void setgrent(void);

DESCRIPTION
Refer to endgrent().
sethostent()

NAME
sethostent — network host database functions

SYNOPSIS
#include <netdb.h>

void sethostent(int stayopen);

DESCRIPTION
Refer to endhostent().
NAME
setitimer — set the value of an interval timer

SYNOPSIS
#include <sys/time.h>

int setitimer(int which, const struct itimerval *restrict value,
              struct itimerval *restrict ovalue);

DESCRIPTION
Refer to getitimer().
setjmp() — set jump point for a non-local goto

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.

A call to setjmp() shall save the calling environment in its env argument for later use by longjmp().

It is unspecified whether setjmp() 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 the name setjmp, the behavior is undefined.

An application shall ensure that an invocation of setjmp() appears in one of the following contexts only:

- The entire controlling expression of a selection or iteration statement
- One operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement
- The operand of a unary '!' operator with the resulting expression being the entire controlling expression of a selection or iteration
- The entire expression of an expression statement (possibly cast to void)

If the invocation appears in any other context, the behavior is undefined.

If the return is from a direct invocation, setjmp() shall return 0. If the return is from a call to longjmp(), setjmp() shall return a non-zero value.

No errors are defined.

In general, sigsetjmp() is more useful in dealing with errors and interrupts encountered in a low-level subroutine of a program.

None.

None.

None.

longjmp(), sigsetjmp(), the Base Definitions volume of IEEE Std 1003.1-2001, <setjmp.h>
**setjmp()**

**CHANGE HISTORY**

First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 6**

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
setkey — set encoding key (CRYPT)

SYNOPSIS
#include <stdlib.h>

void setkey(const char *key);

DESCRIPTION
The setkey() function provides access to an implementation-defined encoding algorithm. The
argument of setkey() is an array of length 64 bytes containing only the bytes with numerical
value of 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is
ignored; this gives a 56-bit key which is used by the algorithm. This is the key that shall be used
with the algorithm to encode a string block passed to encrypt().

The setkey() function shall not change the setting of errno if successful. An application wishing to
check for error situations should set errno to 0 before calling setkey(). If errno is non-zero on
return, an error has occurred.

The setkey() function need not be reentrant. A function that is not required to be reentrant is not
required to be thread-safe.

RETURN VALUE
No values are returned.

ERRORS
The setkey() function shall fail if:

[ENOSYS] The functionality is not supported on this implementation.

EXAMPLES
None.

APPLICATION USAGE
Decoding need not be implemented in all environments. This is related to government
restrictions in some countries on encryption and decryption routines. Historical practice has
been to ship a different version of the encryption library without the decryption feature in the
routines supplied. Thus the exported version of encrypt() does encoding but not decoding.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
crypt(), encrypt(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate that errno is not changed if the function is successful.
NAME
setlocale — set program locale

SYNOPSIS
#include <locale.h>

char *setlocale(int category, const char *locale);

DESCRIPTION
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 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:

- LC_COLLATE: Affects the behavior of regular expressions and the collation functions.
- LC_CTYPE: Affects the behavior of regular expressions, character classification, character conversion functions, and wide-character functions.
- LC_MESSAGES: Affects what strings are expected by commands and utilities as affirmative or negative responses.
- LC_MONETARY: Affects the behavior of functions that handle monetary values.
- LC_NUMERIC: Affects the behavior of functions that handle numeric values.
- LC_TIME: Affects the behavior of the time conversion functions.

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:

- "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().
- "C": Equivalent to "POSIX".
- "": Specifies an implementation-defined native environment. This corresponds to 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.
- A null pointer: Used to direct setlocale() to query the current internationalized environment and return the name of the locale().

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()`.

**ERRORS**

No errors are defined.

**EXAMPLES**

None.

**APPLICATION USAGE**

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:

```c
setlocale(LC_ALL, "De");
setlocale(LC_COLLATE, "Fr@dict");
```

Internationalized programs must call `setlocale()` to initiate a specific language operation. This can be done by calling `setlocale()` as follows:

```c
setlocale(LC_ALL, "");
```

Changing the setting of `LC_MESSAGES` has no effect on catalogs that have already been opened by calls to `catopen()`.

**RATIONALE**

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:

1. Character handling
2. Collating
3. Date/time formatting
4. Numeric editing
5. Monetary formatting
6. Messaging

The `setlocale()` function provides the application developer with the ability to set all or portions, called *categories*, of the international environment. These categories correspond to the areas of functionality mentioned above. The syntax for `setlocale()` is as follows:

```c
char *setlocale(int category, const char *locale);
```

where `category` is the name of one of following categories, namely:

```c
LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME
```
In addition, a special value called \texttt{LC\_ALL} directs \texttt{setlocale()} to set all categories.

There are two primary uses of \texttt{setlocale()}:

1. Querying the international environment to find out what it is set to
2. Setting the international environment, or \texttt{locale}, to a specific value

The behavior of \texttt{setlocale()} in these two areas is described below. Since it is difficult to describe the behavior in words, examples are used to illustrate the behavior of specific uses.

To query the international environment, \texttt{setlocale()} is invoked with a specific category and the NULL pointer as the locale. The NULL pointer is a special directive to \texttt{setlocale()} that tells it to query rather than set the international environment. The following syntax is used to query the name of the international environment:

\begin{verbatim}
setlocale((LC\_ALL, LC\_COLLATE, LC\_CTYPE, LC\_MESSAGES, LC\_MONETARY, \ 
LC\_NUMERIC, LC\_TIME), (char *) NULL);
\end{verbatim}

The \texttt{setlocale()} function shall return the string corresponding to the current international environment. This value may be used by a subsequent call to \texttt{setlocale()} to reset the international environment to this value. However, it should be noted that the return value from \texttt{setlocale()} may be a pointer to a static area within the function and is not guaranteed to remain unchanged (that is, it may be modified by a subsequent call to \texttt{setlocale'}). Therefore, if the purpose of calling \texttt{setlocale()} is to save the value of the current international environment so it can be changed and reset later, the return value should be copied to an array of \texttt{char} in the calling program.

There are three ways to set the international environment with \texttt{setlocale()}:

\texttt{setlocale(category, string)}

This usage sets a specific \texttt{category} in the international environment to a specific value corresponding to the value of the \texttt{string}. A specific example is provided below:

\begin{verbatim}
setlocale(LC\_ALL, "fr\_FR.ISO-8859-1");
\end{verbatim}

In this example, all categories of the international environment are set to the locale corresponding to the string "fr\_FR.ISO-8859-1", or to the French language as spoken in France using the ISO/IEC 8859-1:1998 standard codeset.

If the string does not correspond to a valid locale, \texttt{setlocale()} shall return a NULL pointer and the international environment is not changed. Otherwise, \texttt{setlocale()} shall return the name of the locale just set.

\texttt{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.

\texttt{setlocale(category, "")}

This sets a specific category to an implementation-defined default. This corresponds to the value of the environment variables.

FUTURE DIRECTIONS

None.

SEE ALSO

\texttt{exec, isalnum(), isalpha(), isblank(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), iswalpha(), iswcntrl(), iswctype(), iswdigit(), iswgraph(), iswlower(), iswpunct(), iswspace(), iswupper(), isxdigit(), isxdigit()}
localeconv(), mblen(), mbstowcs(), mbtowc(), nl_langinfo(), printf(), scanf(), setlocale(), strcoll(), strerror(), strftime(), strtof(), tolower(), toupper(), towlower(), towupper(), wcscoll(), wcscat(), wcstombs(), wcsxfrm(), wctomb(), the Base Definitions volume of IEEE Std 1003.1-2001, <langinfo.h>, <locale.h>

40321 Issue 5
The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

40322 Issue 6
Extensions beyond the ISO C standard are marked.

40323 The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
setlogmask — set the log priority mask

SYNOPSIS
#include <syslog.h>

int setlogmask(int maskpri);

DESCRIPTION
Refer to closelog().
NAME
setnetent — network database function

SYNOPSIS
#include <netdb.h>
void setnetent(int stayopen);

DESCRIPTION
Refer to endnetent().
**NAME**

setpgid — set process group ID for job control

**SYNOPSIS**

```c
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

**DESCRIPTION**

The `setpgid()` function shall either join an existing process group or create a new process group within the session of the calling process. The process group ID of a session leader shall not change. Upon successful completion, the process group ID of the process with a process ID that matches `pid` shall be set to `pgid`. As a special case, if `pid` is 0, the process ID of the calling process shall be used. Also, if `pgid` is 0, the process group ID of the indicated process shall be used.

**RETURN VALUE**

Upon successful completion, `setpgid()` shall return 0; otherwise, -1 shall be returned and `errno` shall be set to indicate the error.

**ERRORS**

The `setpgid()` function shall fail if:

- **[EACCES]** The value of the `pid` argument matches the process ID of a child process of the calling process and the child process has successfully executed one of the `exec` functions.
- **[EINVAL]** The value of the `pgid` argument is less than 0, or is not a value supported by the implementation.
- **[EPERM]** The process indicated by the `pid` argument is a session leader.
- **[EPERM]** The value of the `pid` 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.
- **[EPERM]** The value of the `pgid` argument is valid but does not match the process ID of the process indicated by the `pid` argument and there is no process with a process group ID that matches the value of the `pgid` argument in the same session as the calling process.
- **[ESRCH]** The value of the `pid` argument does not match the process ID of the calling process or of a child process of the calling process.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

The `setpgid()` function shall group processes together for the purpose of signaling, placement in foreground or background, and other job control actions.

The `setpgid()` function is similar to the `setpgrp()` function of 4.2 BSD, except that 4.2 BSD allowed the specified new process group to assume any value. This presents certain security problems and is more flexible than necessary to support job control.

To provide tighter security, `setpgid()` only allows the calling process to join a process group already in use inside its session or create a new process group whose process group ID was equal to its process ID.
When a job control shell spawns a new job, the processes in the job must be placed into a new process group via `setpgid()`. There are two timing constraints involved in this action:

1. The new process must be placed in the new process group before the appropriate program is launched via one of the `exec` functions.
2. The new process must be placed in the new process group before the shell can correctly send signals to the new process group.

To address these constraints, the following actions are performed. The new processes call `setpgid()` to alter their own process groups after `fork()` but before `exec`. This satisfies the first constraint. Under 4.3 BSD, the second constraint is satisfied by the synchronization property of `vfork()`; that is, the shell is suspended until the child has completed the `exec`, thus ensuring that the child has completed the `setpgid()`. A new version of `fork()` with this same synchronization property was considered, but it was decided instead to merely allow the parent shell process to adjust the process group of its child processes via `setpgid()`. Both timing constraints are now satisfied by having both the parent shell and the child attempt to adjust the process group of the child process; it does not matter which succeeds first.

Since it would be confusing to an application to have its process group change after it began executing (that is, after `exec`), and because the child process would already have adjusted its process group before this, the `[EACCES]` error was added to disallow this.

One non-obvious use of `setpgid()` is to allow a job control shell to return itself to its original process group (the one in effect when the job control shell was executed). A job control shell does this before returning control back to its parent when it is terminating or suspending itself as a way of restoring its job control “state” back to what its parent would expect. (Note that the original process group of the job control shell typically matches the process group of its parent, but this is not necessarily always the case.)

FUTURE DIRECTIONS

None.

SEE ALSO

`exec`, `getpgrp()`, `setsid()`, `tcsetpgrp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/types.h>`, `<unistd.h>`

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

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.
- The `setpgid()` function is mandatory since `_POSIX_JOB_CONTROL` is required to be defined in this issue. This is a FIPS requirement.
NAME
 setpgrp — set the process group ID

SYNOPSIS
XSI

```c
#include <unistd.h>

pid_t setpgrp(void);
```

DESCRIPTION
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.

The setpgrp() function has no effect when the calling process is a session leader.

RETURN VALUE
Upon completion, setpgrp() shall return the process group ID.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
exec, fork(), getpid(), getsid(), kill(), setpgid(), setsid(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
NAME
setpriority — set the nice value

SYNOPSIS
XSI
#include <sys/resource.h>

int setpriority(int which, id_t who, int nice);

DESCRIPTION
Refer to getpriority().
NAME
setprotoent — network protocol database functions

SYNOPSIS
#include <netdb.h>
void setprotoent(int stayopen);

DESCRIPTION
Refer to endprotoent().
NAME
setpwent — user database function

SYNOPSIS
XSI
#include <pwd.h>

DESCRIPTION
Refer to endpwent().
**NAME**

setregid — set real and effective group IDs

**SYNOPSIS**

```c
#include <unistd.h>

int setregid(gid_t rgid, gid_t egid);
```

**DESCRIPTION**

The `setregid()` function shall set the real and effective group IDs of the calling process.

If `rgid` is −1, the real group ID shall not be changed; if `egid` is −1, the effective group ID shall not be changed.

The real and effective group IDs may be set to different values in the same call.

Only a process with appropriate privileges can set the real group ID and the effective group ID to any valid value.

A non-privileged process can set either the real group ID to the saved set-group-ID from one of the `exec` family of functions, or the effective group ID to the saved set-group-ID or the real group ID.

Any supplementary group IDs of the calling process remain unchanged.

**RETURN VALUE**

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and `errno` set to indicate the error, and neither of the group IDs are changed.

**ERRORS**

The `setregid()` function shall fail if:

- `[EINVAL]` The value of the `rgid` or `egid` argument is invalid or out-of-range.
- `[EPERM]` The process does not have appropriate privileges and a change other than changing the real group ID to the saved set-group-ID, or changing the effective group ID to the real group ID or the saved set-group-ID, was requested.

**EXAMPLES**

None.

**APPLICATION USAGE**

If a set-group-ID process sets its effective group ID to its real group ID, it can still set its effective group ID back to the saved set-group-ID.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`exec`, `setegid()`, `seteuid()`, `getegid()`, `geteuid()`, `setgid()`, `setuid()`, `setreuid()`, `setreuid()`, `setuid()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<unistd.h>`

**CHANGE HISTORY**

First released in Issue 4, Version 2.
setregid()

### Issue 5

- Moved from X/OPEN UNIX extension to BASE.
- 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()*.
NAME
setreuid — set real and effective user IDs

SYNOPSIS
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);

DESCRIPTION
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.

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.

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.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to
indicate the error.

ERRORS
The setreuid() function shall fail if:

EINVAL] The value of the ruid or euid argument is invalid or out-of-range.
[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.

EXAMPLES
Setting the Effective User ID to the Real User ID
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.

#include <unistd.h>
#include <sys/types.h>
... setreuid(getuid(), getuid());
...

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO
getegid(), geteuid(), getgid(), getuid(), setegid(), seteuid(), setgid(), setregid(), setuid(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 4, Version 2.
Moved from X/OPEN UNIX extension to BASE.
NAME
setrlimit — control maximum resource consumption

SYNOPSIS
#include <sys/resource.h>

int setrlimit(int resource, const struct rlimit *rlp);

DESCRIPTION
Refer to getrlimit().
NAME

setservent — network services database functions

SYNOPSIS

#include <netdb.h>

void setservent(int stayopen);

DESCRIPTION

Refer to endservent().
NAME
setsid — create session and set process group ID

SYNOPSIS
#include <unistd.h>

pid_t setsid(void);

DESCRIPTION
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.

RETURN VALUE
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.

ERRORS
The setsid() function shall fail if:

[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.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
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.

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.

FUTURE DIRECTIONS
None.

SEE ALSO
getsid(), setpgid(), setpgrp(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>
<table>
<thead>
<tr>
<th>CHANGE HISTORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.</td>
</tr>
<tr>
<td>Issue 6</td>
</tr>
<tr>
<td>In the SYNOPSIS, the optional include of the <code>&lt;sys/types.h&gt;</code> header is removed.</td>
</tr>
<tr>
<td>The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:</td>
</tr>
<tr>
<td>- The requirement to include <code>&lt;sys/types.h&gt;</code> has been removed. Although <code>&lt;sys/types.h&gt;</code> was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.</td>
</tr>
</tbody>
</table>
 NAME  
 setsockopt — set the socket options

 SYNOPSIS  
 #include <sys/socket.h>

 int setsockopt(int socket, int level, int option_name,
       const void *option_value, socklen_t option_len);

 DESCRIPTION  
 The setsockopt() function shall set the option specified by the option_name argument, at the
 protocol level specified by the level argument, to the value pointed to by the option_value
 argument for the socket associated with the file descriptor specified by the socket argument.

 The level argument specifies the protocol level at which the option resides. To set options at the
 socket level, specify the level argument as SOL_SOCKET. To set 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 (Transport Control Protocol), set level to IPPROTO_TCP
 as defined in the <netinet/in.h> header.

 The option_name argument specifies a single option to set. The option_name argument and any
 specified options are passed uninterpreted to the appropriate protocol module for
 interpretations. The <sys/socket.h> header defines the socket-level options. The options are as
 follows:

 SO_DEBUG  
 Turns on recording of debugging information. This option enables or
 disables debugging in the underlying protocol modules. This option takes
 an int value. This is a Boolean option.

 SO_BROADCAST  
 Permits sending of broadcast messages, if this is supported by the
 protocol. This option takes an int value. This is a Boolean option.

 SO_REUSEADDR  
 Specifies that the rules used in validating addresses supplied to bind() should allow reuse of local addresses, if this is supported by the protocol.
 This option takes an int value. This is a Boolean option.

 SO_KEEPALIVE  
 Keeps connections active by enabling the periodic transmission of
 messages, if this is supported by the protocol. This option takes an int value.
 If the connected socket fails to respond to these messages, the connection
 is broken and threads writing to that socket are notified with a SIGPIPE
 signal. This is a Boolean option.

 SO_LINGER  
 Lingers on a close() if data is present. This option controls the action
 taken when unsent messages queue on a socket and close() is performed.
 If SO_LINGER is set, the system shall block the process during close() until it can transmit the data or until the time expires. If SO_LINGER is not specified, and close() is issued, the system handles the call in a way that allows the process to continue as quickly as possible. This option takes a linger structure, as defined in the <sys/socket.h> header, to specify the state of the option and linger interval.

 SO_OOBINLINE  
 Leaves received out-of-band data (data marked urgent) inline. This
 option takes an int value. This is a Boolean option.

 SO_SNDBUF  
 Sets send buffer size. This option takes an int value.
setsockopt()  

System Interfaces

40680  SO_RCVBUF  Sets receive buffer size. This option takes an int value.

40681  SO_DONTROUTE  Requests that outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option takes an int value. This is a Boolean option.

40686  SO_RCVLOWAT  Sets the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned; for example, out-of-band data.) This option takes an int value. Note that not all implementations allow this option to be set.

40694  SO_RCVTIMEO  Sets the timeout value that specifies the maximum amount of time an input function waits until it completes. It accepts a timeval structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it shall return with a partial count or_errno 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.

40704  SO_SNDLOWAT  Sets the minimum number of bytes to process for socket output operations. Non-blocking output operations shall process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option takes an int value. Note that not all implementations allow this option to be set.

40709  SO_SNDTIMEO  Sets the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it shall return with a partial count or with_errno 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.

40716  For Boolean options, 0 indicates that the option is disabled and 1 indicates that the option is enabled.

40718  Options at other protocol levels vary in format and name.

40719  RETURN VALUE

40720  Upon successful completion, setsockopt() shall return 0. Otherwise, −1 shall be returned and_errno set to indicate the error.

40722  ERRORS

40723  The setsockopt() function shall fail if:

40724  [EBADF]  The socket argument is not a valid file descriptor.

40725  [EDOM]  The send and receive timeout values are too big to fit into the timeout fields in the socket structure.
[EINVAL] The specified option is invalid at the specified socket level or the socket has been shut down.

[EISCONN] The socket is already connected, and a specified option cannot be set while the socket is connected.

[ENO_PROTOOPT] The option is not supported by the protocol.

[ENOTSOCK] The socket argument does not refer to a socket.

[ENOMEM] There was insufficient memory available for the operation to complete.

[ENOBUFS] Insufficient resources are available in the system to complete the call.

EXAMPLES
None.

APPLICATION USAGE
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().

Options may exist at multiple protocol levels. The SO_ options are always present at the uppermost socket level.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.10 (on page 58), bind(), endprotoent(), getsockopt(), socket(), the Base Definitions volume of IEEE Std 1003.1-2001, <netinet/in.h>, <sys/socket.h>

CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
setstate — switch pseudo-random number generator state arrays

SYNOPSIS
XSI
#include <stdlib.h>

char *setstate(const char *state);

DESCRIPTION
Refer to initstate().
**NAME**
setuid — set user ID

**SYNOPSIS**
```c
#include <unistd.h>

int setuid(uid_t uid);
```

**DESCRIPTION**
If the process has appropriate privileges, `setuid()` shall set the real user ID, effective user ID, and saved set-user-ID of the calling process to `uid`.

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.

The `setuid()` function shall not affect the supplementary group list in any way.

**RETURN VALUE**
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and `errno` set to indicate the error.

**ERRORS**
The `setuid()` function shall fail, return −1, and set `errno` to the corresponding value if one or more of the following are true:

- `[EINVAL]` The value of the `uid` argument is invalid and not supported by the implementation.
- `[EPERM]` The process does not have appropriate privileges and `uid` does not match the real user ID or the saved set-user-ID.

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
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.

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
setuid() scope of IEEE Std 1003.1-2001. These utilities wish to change the user ID irrevocably to a new value, generally that of an unprivileged user. The latter use is needed for conforming applications that are installed with the set-user-ID bit and need to perform operations using the real user ID.

IEEE Std 1003.1-2001 augments the latter functionality with a mandatory feature named _POSIX_SAVED_IDS. This feature permits a set-user-ID application to switch its effective user ID back and forth between the values of its exec-time real user ID and effective user ID. Unfortunately, the POSIX.1-1990 standard did not permit a conforming application using this feature to work properly when it happened to be executed with the (implementation-defined) appropriate privilege. Furthermore, the application did not even have a means to tell whether it had this privilege. Since the saved set-user-ID feature is quite desirable for applications, as evidenced by the fact that NIST required it in FIPS 151-2, it has been mandated by IEEE Std 1003.1-2001. However, there are implementors who have been reluctant to support it given the limitation described above.

The 4.3BSD system handles the problem by supporting separate functions: setuid() (which always sets both the real and effective user IDs, like setuid() in IEEE Std 1003.1-2001 for privileged users), and seteuid() (which always sets just the effective user ID, like setuid() in IEEE Std 1003.1-2001 for non-privileged users). This separation of functionality into distinct functions seems desirable. 4.3BSD does not support the saved set-user-ID feature. It supports similar functionality of switching the effective user ID back and forth via setreuid(), which permits reversing the real and effective user IDs. This model seems less desirable than the saved set-user-ID because the real user ID changes as a side effect. The current 4.4BSD includes saved effective IDs and uses them for seteuid() and setegid() as described above. The setreuid() and setregid() functions will be deprecated or removed.

The solution here is:

- Require that all implementations support the functionality of the saved set-user-ID, which is set by the exec functions and by privileged calls to setuid().
- Add the seteuid() and setegid() functions as portable alternatives to setuid() and setgid() for non-privileged and privileged processes.

Historical systems have provided two mechanisms for a set-user-ID process to change its effective user ID to be the same as its real user ID in such a way that it could return to the original effective user ID: the use of the setuid() function in the presence of a saved set-user-ID, or the use of the BSD setreuid() function, which was able to swap the real and effective user IDs. The changes included in IEEE Std 1003.1-2001 provide a new mechanism using seteuid() in conjunction with a saved set-user-ID. Thus, all implementations with the new setuid() mechanism will have a saved set-user-ID for each process, and most of the behavior controlled by _POSIX_SAVED_IDS has been changed to agree with the case where the option was defined.

The kill() function is an exception. Implementors of the new setuid() mechanism will generally be required to maintain compatibility with the older mechanisms previously supported by their systems. However, compatibility with this use of setreuid() and with the _POSIX_SAVED_IDS behavior of kill() is unfortunately complicated. If an implementation with a saved set-user-ID allows a process to use setreuid() to swap its real and effective user IDs, but were to leave the saved set-user-ID unmodified, the process would then have an effective user ID equal to the original real user ID, and both real and saved set-user-ID would be equal to the original effective user ID. In that state, the real user would be unable to kill the process, even though the effective user ID of the process matches that of the real user, if the kill() behavior of _POSIX_SAVED_IDS was used. This is obviously not acceptable. The alternative choice, which is used in at least one implementation, is to change the saved set-user-ID to the effective user ID during most calls to setreuid(). The standard developers considered that alternative to be less correct than the
retention of the old behavior of kill() in such systems. Current conforming applications shall accommodate either behavior from kill(), and there appears to be no strong reason for kill() to check the saved set-user-ID rather than the effective user ID.

FUTURE DIRECTIONS
None.

SEE ALSO
exec, getegid(), geteuid(), getgid(), getuid(), setegid(), seteuid(), setgid(), setregid(), setreuid(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

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.

• The functionality associated with _POSIX_SAVED_IDS is now mandatory. This is a FIPS requirement.

The following changes were made to align with the IEEE P1003.1a draft standard:

• The effects of setuid() in processes without appropriate privileges are changed.

• A requirement that the supplementary group list is not affected is added.
NAME
setutxent — reset the user accounting database to the first entry

SYNOPSIS
XSI
#include <utmpx.h>
void setutxent(void);

DESCRIPTION
Refer to endutxent().
NAME
setvbuf — assign buffering to a stream

SYNOPSIS
#include <stdio.h>

int setvbuf(FILE *restrict stream, char *restrict buf, int type,
size_t size);

DESCRIPTION
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

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.

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.

For information about streams, see Section 2.5 (on page 34).

RETURN VALUE
Upon successful completion, setvbuf() shall return 0. Otherwise, it shall return a non-zero value
if an invalid value is given for type or if the request cannot be honored, and may set errno to
indicate the error.

ERRORS
The setvbuf() function may fail if:

- [EBADF] The file descriptor underlying stream is not valid.

EXAMPLES
None.

APPLICATION USAGE
A common source of error is allocating buffer space as an “automatic” variable in a code block,
and then failing to close the stream in the same block.

With setvbuf(), allocating a buffer of size bytes does not necessarily imply that all of size bytes are
used for the buffer area.

Applications should note that many implementations only provide line buffering on input from
terminal devices.

RATIONALE
None.
FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.5 (on page 34), fopen(), setbuf(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are marked.

The setvbuf() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
shm_open — open a shared memory object (REALTIME)

SYNOPSIS
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);

DESCRIPTION
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.

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.

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:

- O_RDONLY Open for read access only.
- O_RDWR Open for read or write access.

Any combination of the remaining flags may be specified in the value of oflag:

- 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.

- 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.
O_TRUNC

If the shared memory object exists, and it is successfully opened O_RDWR, the object shall be truncated to zero length and the mode and owner shall be unchanged by this function call. The result of using O_TRUNC with O_RDONLY is undefined.

When a shared memory object is created, the state of the shared memory object, including all data associated with the shared memory object, persists until the shared memory object is unlinked and all other references are gone. It is unspecified whether the name and shared memory object state remain valid after a system reboot.

RETURN VALUE

Upon successful completion, the `shm_open()` function shall return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it shall return −1 and set `errno` to indicate the error.

ERRORS

The `shm_open()` function shall fail if:

- [EACCES] The shared memory object exists and the permissions specified by `oflag` are denied, or the shared memory object does not exist and permission to create the shared memory object is denied, or O_TRUNC is specified and write permission is denied.
- [EEXIST] O_CREAT and O_EXCL are set and the named shared memory object already exists.
- [EINVAL] The `shm_open()` operation is not supported for the given name.
- [EMFILE] Too many file descriptors are currently in use by this process.
- [ENAMETOOLONG] The length of the `name` argument exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.
- [ENFILE] Too many shared memory objects are currently open in the system.
- [ENOENT] O_CREAT is not set and the named shared memory object does not exist.
- [ENOSPC] There is insufficient space for the creation of the new shared memory object.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

When the Memory Mapped Files option is supported, the normal `open()` call is used to obtain a descriptor to a file to be mapped according to existing practice with `mmap()`. When the Shared Memory Objects option is supported, the `shm_open()` function shall obtain a descriptor to the shared memory object to be mapped.

There is ample precedent for having a file descriptor represent several types of objects. In the POSIX.1-1990 standard, a file descriptor can represent a file, a pipe, a FIFO, a tty, or a directory. Many implementations simply have an operations vector, which is indexed by the file descriptor type and does very different operations. Note that in some cases the file descriptor passed to generic operations on file descriptors is returned by `open()` or `creat()` and in some cases returned by alternate functions, such as `pipe()`. The latter technique is used by `shm_open()`.
Note that such shared memory objects can actually be implemented as mapped files. In both cases, the size can be set after the open using `ftruncate()`. The `shm_open()` function itself does not create a shared object of a specified size because this would duplicate an extant function that sets the size of an object referenced by a file descriptor.

On implementations where memory objects are implemented using the existing file system, the `shm_open()` function may be implemented using a macro that invokes `open()`, and the `shm_unlink()` function may be implemented using a macro that invokes `unlink()`.

For implementations without a permanent file system, the definition of the name of the memory objects is allowed not to survive a system reboot. Note that this allows systems with a permanent file system to implement memory objects as data structures internal to the implementation as well.

On implementations that choose to implement memory objects using memory directly, a `shm_open()` followed by an `ftruncate()` and `close()` can be used to preallocate a shared memory area and to set the size of that preallocation. This may be necessary for systems without virtual memory hardware support in order to ensure that the memory is contiguous.

The set of valid open flags to `shm_open()` was restricted to `O_RDONLY`, `O_RDWR`, `O_CREAT`, and `O_TRUNC` because these could be easily implemented on most memory mapping systems. This volume of IEEE Std 1003.1-2001 is silent on the results if the implementation cannot supply the requested file access because of implementation-defined reasons, including hardware ones.

The error conditions `[EACCES]` and `[ENOTSUP]` are provided to inform the application that the implementation cannot complete a request.

[EACCES] indicates for implementation-defined reasons, probably hardware-related, that the implementation cannot comply with a requested mode because it conflicts with another requested mode. An example might be that an application desires to open a memory object two times, mapping different areas with different access modes. If the implementation cannot map a single area into a process space in two places, which would be required if different access modes were required for the two areas, then the implementation may inform the application at the time of the second open.

[ENOTSUP] indicates for implementation-defined reasons, probably hardware-related, that the implementation cannot comply with a requested mode at all. An example would be that the hardware of the implementation cannot support write-only shared memory areas.

On all implementations, it may be desirable to restrict the location of the memory objects to specific file systems for performance (such as a RAM disk) or implementation-defined reasons (shared memory supported directly only on certain file systems). The `shm_open()` function may be used to enforce these restrictions. There are a number of methods available to the application to determine an appropriate name of the file or the location of an appropriate directory. One way is from the environment via `getenv()`. Another would be from a configuration file.

This volume of IEEE Std 1003.1-2001 specifies that memory objects have initial contents of zero when created. This is consistent with current behavior for both files and newly allocated memory. For those implementations that use physical memory, it would be possible that such implementations could simply use available memory and give it to the process uninitialized. This, however, is not consistent with standard behavior for the uninitialized data area, the stack, and of course, files. Finally, it is highly desirable to set the allocated memory to zero for security reasons. Thus, initializing memory objects to zero is required.
shm_open()

FUTURE DIRECTIONS
None.

SEE ALSO
close(), dup(), exec, fcntl(), mmap(), shmat(), shmctl(), shmdt(), shm_unlink(), umask(), the Base Definitions volume of IEEE Std 1003.1-2001, <fcntl.h>, <sys/mman.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6
The shm_open() function is marked as part of the Shared Memory Objects option.
The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Shared Memory Objects option.
NAME
shm_unlink — remove a shared memory object (REALTIME)

SYNOPSIS
#include <sys/mman.h>

int shm_unlink(const char *name);

DESCRIPTION
The shm_unlink() function shall remove the name of the shared memory object named by the
string pointed to by name.

If one or more references to the shared memory object exist when the object is unlinked, the
name shall be removed before shm_unlink() returns, but the removal of the memory object
contents shall be postponed until all open and map references to the shared memory object have
been removed.

Even if the object continues to exist after the last shm_unlink(), reuse of the name shall
subsequently cause shm_open() to behave as if no shared memory object of this name exists (that
is, shm_open() will fail if O_CREAT is not set, or will create a new shared memory object if
O_CREAT is set).

RETURN VALUE
Upon successful completion, a value of zero shall be returned. Otherwise, a value of −1 shall be
returned and errno set to indicate the error. If −1 is returned, the named shared memory object
shall not be changed by this function call.

ERRORS
The shm_unlink() function shall fail if:

[EACCES]  Permission is denied to unlink the named shared memory object.

[ENAMETOOLONG]
The length of the name argument exceeds PATH_MAX or a pathname
component is longer than NAME_MAX.

[ENOENT]  The named shared memory object does not exist.

EXAMPLES
None.

APPLICATION USAGE
Names of memory objects that were allocated with open() are deleted with unlink() in the usual
fashion. Names of memory objects that were allocated with shm_open() are deleted with
shm_unlink(). Note that the actual memory object is not destroyed until the last close and
unmap on it have occurred if it was already in use.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
close(), mmap(), munmap(), shmat(), shmctl(), shmdt(), shm_open(), the Base Definitions volume
of IEEE Std 1003.1-2001, <sys/mman.h>
CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The `shm_unlink()` function is marked as part of the Shared Memory Objects option.

In the DESCRIPTION, text is added to clarify that reusing the same name after a `shm_unlink()`
will not attach to the old shared memory object.

The [ENOSYS] error condition has been removed as stubs need not be provided if an
implementation does not support the Shared Memory Objects option.
NAME
shmat — XSI shared memory attach operation

SYNOPSIS
XSI
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);

DESCRIPTION
The shmat() function operates on XSI shared memory (see the Base Definitions volume of
IEEE Std 1003.1-2001, Section 3.340, Shared Memory Object). It is unspecified whether this
function interoperates with the realtime interprocess communication facilities defined in Section
2.8 (on page 41).

The shmat() function attaches the shared memory segment associated with the shared memory
identifier specified by shmid to the address space of the calling process. The segment is attached
at the address specified by one of the following criteria:

• If shmaddr is a null pointer, the segment is attached at the first available address as selected
  by the system.

• If shmaddr is not a null pointer and (shmflg &SHM_RND) is non-zero, the segment is attached
  at the address given by (shmaddr −((uintptr_t)shmaddr %SHMLBA)). The character ‘%’ is the
  C-language remainder operator.

• If shmaddr is not a null pointer and (shmflg &SHM_RND) is 0, the segment is attached at the
  address given by shmaddr.

• The segment is attached for reading if (shmflg &SHM_RDONLY) is non-zero and the calling
  process has read permission; otherwise, if it is 0 and the calling process has read and write
  permission, the segment is attached for reading and writing.

RETURN VALUE
Upon successful completion, shmat() shall increment the value of shm_nattach in the data
structure associated with the shared memory ID of the attached shared memory segment and
return the segment’s start address.

Otherwise, the shared memory segment shall not be attached, shmat() shall return −1, and errno
shall be set to indicate the error.

ERRORS
The shmat() function shall fail if:

[EACCES] Operation permission is denied to the calling process; see Section 2.7 (on page
39).

[EINVAL] The value of shmid is not a valid shared memory identifier, the shmaddr is not a
null pointer, and the value of (shmaddr −((uintptr_t)shmaddr %SHMLBA)) is an
illegal address for attaching shared memory; or the shmaddr is not a null
pointer, (shmflg &SHM_RND) is 0, and the value of shmaddr is an illegal
address for attaching shared memory.

[EMFILE] The number of shared memory segments attached to the calling process
would exceed the system-imposed limit.

[ENOMEM] The available data space is not large enough to accommodate the shared
memory segment.
EXAMPLES
None.

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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.7 (on page 39), Section 2.8 (on page 41), exec, exit(), fork(), shmtctl(), shmdt(), shmget(), shm_open(), shm_unlink(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/shm.h>

CHANGE HISTORY
First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5
Moved from SHARED MEMORY to BASE.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

Issue 6
The Open Group Corrigendum U021/13 is applied.
NAME

shmctl — XSI shared memory control operations

SYNOPSIS

XSI
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

DESCRIPTION

The shmctl() function operates on XSI shared memory (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.340, Shared Memory Object). It is unspecified whether this function interoperares with the realtime interprocess communication facilities defined in Section 2.8 (on page 41).

The shmctl() function provides a variety of shared memory control operations as specified by cmd. The following values for cmd are available:

IPC_STAT Place the current value of each member of the shmid_ds data structure associated with shmid into the structure pointed to by buf. The contents of the structure are defined in <sys/shm.h>.

IPC_SET Set the value of the following members of the shmid_ds data structure associated with shmid to the corresponding value found in the structure pointed to by buf:

shm_perm.uid
shm_perm.gid
shm_perm.mode Low-order nine bits.

IPC_SET 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 shm_perm.uid or shm_perm.cuid in the shmid_ds data structure associated with shmid.

IPC_RMID Remove the shared memory identifier specified by shmid from the system and destroy the shared memory segment and shmid_ds data structure associated with it. IPC_RMID 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 shm_perm.uid or shm_perm.cuid in the shmid_ds data structure associated with shmid.

RETURN VALUE

Upon successful completion, shmctl() shall return 0; otherwise, it shall return −1 and set errno to indicate the error.

ERRORS

The shmctl() function shall fail if:

[EACCES] The argument cmd is equal to IPC_STAT and the calling process does not have read permission; see Section 2.7 (on page 39).

EINVAL The value of shmid is not a valid shared memory identifier, or the value of cmd is not a valid command.

[EPERM] The argument cmd 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 shm_perm.uid or shm_perm.cuid in the data structure associated with shmid.
The `shmctl()` function may fail if:

- [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.

**EXAMPLES**

None.

**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.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

Section 2.7 (on page 39), Section 2.8 (on page 41), `shmat()`, `shmdt()`, `shmget()`, `shm_open()`, `shm_unlink()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<sys/shm.h>`

**CHANGE HISTORY**

- **Issue 5**
  - First released in Issue 2. Derived from Issue 2 of the SVID.
  - Moved from SHARED MEMORY to BASE.
  - The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.
NAME
shmdt — XSI shared memory detach operation

SYNOPSIS
#include <sys/shm.h>

int shmdt(const void *shmaddr);

DESCRIPTION
The shmdt() function operates on XSI shared memory (see the Base Definitions volume of
IEEE Std 1003.1-2001, Section 3.340, Shared Memory Object). It is unspecified whether this
function interoperates with the realtime interprocess communication facilities defined in Section
2.8 (on page 41).

The shmdt() function detaches the shared memory segment located at the address specified by
shmaddr from the address space of the calling process.

RETURN VALUE
Upon successful completion, shmdt() shall decrement the value of shm_nattch in the data
structure associated with the shared memory ID of the attached shared memory segment and
return 0.

Otherwise, the shared memory segment shall not be detached, shmdt() shall return -1, and errno
shall be set to indicate the error.

ERRORS
The shmdt() function shall fail if:

[EINVAL] The value of shmaddr is not the data segment start address of a shared
memory segment.

EXAMPLES
None.

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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.7 (on page 39), Section 2.8 (on page 41), exec, exit(), fork(), shmat(), shmctl(), shmget(),
shm_open(), shm_unlink(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/shm.h>

CHANGE HISTORY
First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5
Moved from SHARED MEMORY to BASE.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE
DIRECTIONS to a new APPLICATION USAGE section.
shmget( )

NAME
shmget — get an XSI shared memory segment

SYNOPSIS
XSI
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);

DESCRIPTION
The shmget( ) function operates on XSI shared memory (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.340, Shared Memory Object). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 41).

The shmget( ) function shall return the shared memory identifier associated with key.

A shared memory identifier, associated data structure, and shared memory segment of at least size bytes (see <sys/shm.h>) are created for key if one of the following is true:

• The argument key is equal to IPC_PRIVATE.
• The argument key does not already have a shared memory identifier associated with it and (shmflg &IPC_CREAT) is non-zero.

Upon creation, the data structure associated with the new shared memory identifier shall be initialized as follows:

• The values of shm_perm.cuid, shm_perm.uid, shm_perm.cgid, and shm_perm.gid are set equal to the effective user ID and effective group ID, respectively, of the calling process.
• The low-order nine bits of shm_perm.mode are set equal to the low-order nine bits of shmflg.
• The value of shm_segsz is set equal to the value of size.
• The values of shm_lpid, shm_nattch, shm_atime, and shm_dtime are set equal to 0.
• The value of shm_ctime is set equal to the current time.

When the shared memory segment is created, it shall be initialized with all zero values.

RETURN VALUE
Upon successful completion, shmget() shall return a non-negative integer, namely a shared memory identifier; otherwise, it shall return −1 and set errno to indicate the error.

ERRORS
The shmget() function shall fail if:

• [EACCES] A shared memory identifier exists for key but operation permission as specified by the low-order nine bits of shmflg would not be granted; see Section 2.7 (on page 39).
• [EEXIST] A shared memory identifier exists for the argument key but (shmflg &IPC_CREAT) &&(shmflg &IPC_EXCL) is non-zero.
• [EINVAL] A shared memory segment is to be created and the value of size is less than the system-imposed minimum or greater than the system-imposed maximum.
• [EINVAL] No shared memory segment is to be created and a shared memory segment exists for key but the size of the segment associated with it is less than size and size is not 0.
A shared memory identifier does not exist for the argument \textit{key} and \texttt{(shmflg &IPC_CREAT)} is 0.

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.

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.

\textbf{EXAMPLES}

None.

\textbf{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.

\textbf{RATIONALE}

None.

\textbf{FUTURE DIRECTIONS}

None.

\textbf{SEE ALSO}

Section 2.7 (on page 39), Section 2.8 (on page 41), \texttt{shmat()}, \texttt{shmctll()}, \texttt{shmdt()}, \texttt{shm_open()}, \texttt{shm_unlink()}, the Base Definitions volume of IEEE Std 1003.1-2001, <\texttt{sys/shm.h}>

First released in Issue 2. Derived from Issue 2 of the SVID.

Moved from SHARED MEMORY to BASE.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.
shutdown()

NAME
shutdown — shut down socket send and receive operations

SYNOPSIS
#include <sys/socket.h>

int shutdown(int socket, int how);

DESCRIPTION
The shutdown() function shall cause all or part of a full-duplex connection on the socket
associated with the file descriptor socket to be shut down.
The shutdown() function takes the following arguments:

socket Specifies the file descriptor of the socket.
how Specifies the type of shutdown. The values are as follows:
    SHUT_RD Disables further receive operations.
    SHUT_WR Disables further send operations.
    SHUT_RDWR Disables further send and receive operations.

The shutdown() function disables subsequent send and/or receive operations on a socket,
depending on the value of the how argument.

RETURN VALUE
Upon successful completion, shutdown() shall return 0; otherwise, −1 shall be returned and
errno set to indicate the error.

ERRORS
The shutdown() function shall fail if:

[EBADF] The socket argument is not a valid file descriptor.
[EINVAL] The how argument is invalid.
[ENOTCONN] The socket is not connected.
[ENOTSOCK] The socket argument does not refer to a socket.
[ENOBUFS] Insufficient resources were available in the system to perform the operation.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
getsockopt(), read(), recv(), recvfrom(), recvmsg(), select(), send(), sendto(), setsockopt(), socket(),
write(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>
CHANGE HISTORY
First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME

sigaction — examine and change a signal action

SYNOPSIS

```c
#include <signal.h>

int sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact);
```

DESCRIPTION

The `sigaction()` function allows the calling process to examine and/or specify the action to be associated with a specific signal. The argument `sig` specifies the signal; acceptable values are defined in `<signal.h>`.

The structure `sigaction`, used to describe an action to be taken, is defined in the `<signal.h>` header to include at least the following members:

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void(*) (int)</td>
<td>sa_handler</td>
<td>SIG_DFL, SIG_IGN, or pointer to a function.</td>
</tr>
<tr>
<td>sigset_t</td>
<td>sa_mask</td>
<td>Additional set of signals to be blocked during execution of signal-catching function.</td>
</tr>
<tr>
<td>int</td>
<td>sa_flags</td>
<td>Special flags to affect behavior of signal.</td>
</tr>
<tr>
<td>void(*)(int, siginfo_t *, void *)</td>
<td>sa_sigaction</td>
<td>Signal-catching function.</td>
</tr>
</tbody>
</table>

The storage occupied by `sa_handler` and `sa_sigaction` may overlap, and a conforming application shall not use both simultaneously.

If the argument `act` is not a null pointer, it points to a structure specifying the action to be associated with the specified signal. If the argument `oact` is not a null pointer, the action previously associated with the signal is stored in the location pointed to by the argument `oact`. If the argument `act` is a null pointer, signal handling is unchanged; thus, the call can be used to enquire about the current handling of a given signal. The SIGKILL and SIGSTOP signals shall not be added to the signal mask using this mechanism; this restriction shall be enforced by the system without causing an error to be indicated.

If the SA_SIGINFO flag (see below) is cleared in the `sa_flags` field of the `sigaction` structure, the `sa_handler` field identifies the action to be associated with the specified signal. If the SA_SIGINFO flag is set in the `sa_flags` field, and the implementation supports the Realtime Signals Extension option or the XSI Extension option, the `sa_sigaction` field specifies a signal-catching function. If the SA_SIGINFO bit is cleared and the `sa_handler` field specifies a signal-catching function, or if the SA_SIGINFO bit is set, the `sa_mask` field identifies a set of signals that shall be added to the signal mask of the thread before the signal-catching function is invoked. If the `sa_handler` field specifies a signal-catching function, the `sa_mask` field identifies a set of signals that shall be added to the process’ signal mask before the signal-catching function is invoked.

The `sa_flags` field can be used to modify the behavior of the specified signal.

The following flags, defined in the `<signal.h>` header, can be set in `sa_flags`:

- `SA_NOCLDSTOP` Do not generate SIGCHLD when children stop or stopped children continue.
![Image of a document](image-url)

If `sig` is `SIGCHLD` and the `SA_NOCLDSTOP` flag is not set in `sa_flags`, and the implementation supports the `SIGCHLD` signal, then a `SIGCHLD` signal shall be generated for the calling process whenever any of its child processes stop and a `SIGCHLD` signal may be generated for the calling process whenever any of its stopped child processes are continued. If `sig` is `SIGCHLD` and the `SA_NOCLDSTOP` flag is set in `sa_flags`, then the implementation shall not generate a `SIGCHLD` signal in this way.

- **SA_ONSTACK**
  - If set and an alternate signal stack has been declared with `sigaltstack()`, the signal shall be delivered to the calling process on that stack. Otherwise, the signal shall be delivered on the current stack.

- **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.

  - **Note:** `SIGILL` and `SIGTRAP` cannot be automatically reset when delivered; the system silently enforces this restriction.

- **SA_RESTART**
  - This flag affects the behavior of interruptible functions; that is, those specified to fail with `errno` 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 `errno` set to `[EINTR]`.

- **SA_SIGINFO**
  - If cleared and the signal is caught, the signal-catching function shall be entered as:

    ```c
    void func(int signo);
    ```

    where `signo` is the only argument to the signal-catching function. In this case, the application shall use the `sa_handler` member to describe the signal-catching function and the application shall not modify the `sa_sigaction` member.

  - If `SA_SIGINFO` is set and the signal is caught, the signal-catching function shall be entered as:

    ```c
    void func(int signo, siginfo_t *info, void *context);
    ```

    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 process’ context that was interrupted when the signal was delivered. In this case, the application shall use the `sa_sigaction` member to describe the signal-catching function and the application shall not modify the `sa_handler` member.

    - The `si_signo` member contains the system-generated signal number.

  - **XSI**
    - The `si_errno` 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.
The \textit{si\_code} member contains a code identifying the cause of the signal. If the value of \textit{si\_code} is less than or equal to 0, then the signal was generated by a process and \textit{si\_pid} and \textit{si\_uid}, respectively, indicate the process ID and the real user ID of the sender. The \texttt{<signal.h>} header description contains information about the signal-specific contents of the elements of the \texttt{siginfo\_t} type.

If set, and \textit{sig} 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 \texttt{wait()}, \texttt{waitid()}, and \texttt{waitpid()} shall fail and set \textit{errno} to [ECHILD]. Otherwise, terminating child processes shall be transformed into zombie processes, unless SIGCHLD is set to SIG\_IGN.

When a signal is caught by a signal-catching function installed by \texttt{sigaction()}, a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either \texttt{sigprocmask()} or \texttt{sigsuspend()} is made). This mask is formed by taking the union of the current signal mask and the value of the \textit{sa\_mask} for the signal being delivered unless \texttt{SA\_NODEFER} or \texttt{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.

Once an action is installed for a specific signal, it shall remain installed until another action is explicitly requested (by another call to \texttt{sigaction()}), until the \texttt{SA\_RESETHAND} flag causes resetting of the handler, or until one of the \texttt{exec} functions is called.

If the previous action for \textit{sig} had been established by \texttt{signal()}, the values of the fields returned in the structure pointed to by \texttt{oact} are unspecified, and in particular \texttt{oact->sa\_handler} is not necessarily the same value passed to \texttt{signal()}. However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to \texttt{sigaction()} via the \texttt{act} argument, handling of the signal shall be as if the original call to \texttt{signal()} were repeated.

If \texttt{sigaction()} fails, no new signal handler is installed.

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 \textit{errno} set to [EINVAL].

If \texttt{SA\_SIGINFO} is not set in \texttt{sa\_flags}, then the disposition of subsequent occurrences of \textit{sig} 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 \texttt{SA\_SIGINFO} is set in \texttt{sa\_flags}, then subsequent occurrences of \textit{sig} generated by \texttt{sigqueue()} or as a result of any signal-generating function that supports the specification of an application-defined value (when \textit{sig} 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 \textit{si\_value} member of the \texttt{siginfo\_t} structure.

The result of the use of \texttt{sigaction()} and a \texttt{sigwait()} function concurrently within a process on the same signal is unspecified.
 Upon successful completion, \texttt{sigaction()} shall return 0; otherwise, \texttt{−1} shall be returned, \texttt{errno} shall be set to indicate the error, and no new signal-catching function shall be installed.

The \texttt{sigaction()} function shall fail if:

- \texttt{EINVAL} The \texttt{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.
- \texttt{ENOTSUP} The \texttt{SA_SIGINFO} bit flag is set in the \texttt{sa_flags} field of the \texttt{sigaction} structure, and the implementation does not support either the Realtime Signals Extension option, or the XSI Extension option.

The \texttt{sigaction()} function may fail if:

- \texttt{EINVAL} An attempt was made to set the action to \texttt{SIG_DFL} for a signal that cannot be caught or ignored (or both).

None.

The \texttt{sigaction()} function supersedes the \texttt{signal()} function, and should be used in preference. In particular, \texttt{sigaction()} and \texttt{signal()} should not be used in the same process to control the same signal. The behavior of reentrant functions, as defined in the DESCRIPTION, is as specified by this volume of IEEE Std 1003.1-2001, regardless of invocation from a signal-catching function. This is the only intended meaning of the statement that reentrant functions may be used in signal-catching functions without restrictions. Applications must still consider all effects of such functions on such things as data structures, files, and process state. In particular, application writers need to consider the restrictions on interactions when interrupting \texttt{sleep()} and interactions among multiple handles for a file description. The fact that any specific function is listed as reentrant does not necessarily mean that invocation of that function from a signal-catching function is recommended.

In order to prevent errors arising from interrupting non-reentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore (see \texttt{semget()}, \texttt{sem_init()}, \texttt{sem_open()}, and so on). Note in particular that even the “safe” functions may modify \texttt{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 \texttt{longjmp()} and \texttt{siglongjmp()} are not in the list of reentrant functions. This is because the code executing after \texttt{longjmp()} and \texttt{siglongjmp()} can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use \texttt{longjmp()} and \texttt{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 \texttt{malloc()} or \texttt{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 \texttt{abort()}, \texttt{kill()}, or \texttt{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 \texttt{volatile sig_atomic_t}. Furthermore, if such a call fails, the value of \texttt{errno} is unspecified.
Usually, the signal is executed on the stack that was in effect before the signal was delivered. An alternate stack may be specified to receive a subset of the signals being caught.

When the signal handler returns, the receiving process resumes execution at the point it was interrupted unless the signal handler makes other arrangements. If longjmp() or _longjmp() is used to leave the signal handler, then the signal mask must be explicitly restored by the process.

This volume of IEEE Std 1003.1-2001 defines the third argument of a signal handling function when SA_SIGINFO is set as a void * instead of a ucontext_t *, but without requiring type checking. New applications should explicitly cast the third argument of the signal handling function to ucontext_t *.

The BSD optional four argument signal handling function is not supported by this volume of IEEE Std 1003.1-2001. The BSD declaration would be:

```c
void handler(int sig, int code, struct sigcontext *scp,
             char *addr);
```

where sig is the signal number, code is additional information on certain signals, scp is a pointer to the sigcontext structure, and addr is additional address information. Much the same information is available in the objects pointed to by the second argument of the signal handler specified when SA_SIGINFO is set.

**RATIONALE**

Although this volume of IEEE Std 1003.1-2001 requires that signals that cannot be ignored shall not be added to the signal mask when a signal-catching function is entered, there is no explicit requirement that subsequent calls to sigaction() reflect this in the information returned in the oact argument. In other words, if SIGKILL is included in the sa_mask field of act, it is unspecified whether or not a subsequent call to sigaction() returns with SIGKILL included in the sa_mask field of oact.

The SA_NOCLDSTOP flag, when supplied in the act->sa_flags parameter, allows overloading SIGCHLD with the System V semantics that each SIGCLD signal indicates a single terminated child. Most conforming applications that catch SIGCHLD are expected to install signal-catching functions that repeatedly call the waitpid() function with the WNOHANG flag set, acting on each child for which status is returned, until waitpid() returns zero. If stopped children are not of interest, the use of the SA_NOCLDSTOP flag can prevent the overhead from invoking the signal-catching routine when they stop.

Some historical implementations also define other mechanisms for stopping processes, such as the ptrace() function. These implementations usually do not generate a SIGCHLD signal when processes stop due to this mechanism; however, that is beyond the scope of this volume of IEEE Std 1003.1-2001.

This volume of IEEE Std 1003.1-2001 requires that calls to sigaction() that supply a NULL act argument succeed, even in the case of signals that cannot be caught or ignored (that is, SIGKILL or SIGSTOP). The System V signal() and BSD sigvec() functions return [EINVAL] in these cases and, in this respect, their behavior varies from sigaction().

This volume of IEEE Std 1003.1-2001 requires that sigaction() properly save and restore a signal action set up by the ISO C standard signal() function. However, there is no guarantee that the reverse is true, nor could there be given the greater amount of information conveyed by the sigaction structure. Because of this, applications should avoid using both functions for the same signal in the same process. Since this cannot always be avoided in case of general-purpose library routines, they should always be implemented with sigaction().

It was intended that the signal() function should be implementable as a library routine using sigaction().
The POSIX Realtime Extension extends the `sigaction()` function as specified by the POSIX.1-1990 standard to allow the application to request on a per-signal basis via an additional signal action flag that the extra parameters, including the application-defined signal value, if any, be passed to the signal-catchign function.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

Section 2.4 (on page 28), `bsd_signal()`, `kill()`, `_longjmp()`, `longjmp()`, `raise()`, `semget()`, `sem_init()`, `sem_open()`, `sigaddset()`, `sigaltstack()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal()`, `sigprocmask()`, `sigsuspend()`, `wait()`, `waitid()`, `waitpid()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`, `<ucontext.h>`

**CHANGE HISTORY**

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

**Issue 5**

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and POSIX Threads Extension.

In the DESCRIPTION, the second argument to `func` when `SA_SIGINFO` is set is no longer permitted to be NULL, and the description of permitted `siginfo_t` contents is expanded by reference to `<signal.h>`.

Since the X/OPEN UNIX Extension functionality is now folded into the BASE, the `[ENOTSUP]` error is deleted.

**Issue 6**

The Open Group Corrigendum U028/7 is applied. In the paragraph entitled “Signal Effects on Other Functions”, a reference to `sigpending()` is added.

In the DESCRIPTION, the text “Signal Generation and Delivery”, “Signal Actions”, and “Signal Effects on Other Functions” are moved to a separate section of this volume of IEEE Std 1003.1-2001.

Text describing functionality from the Realtime Signals option is marked.

The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- The `[ENOTSUP]` error condition is added.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The `restrict` keyword is added to the `sigaction()` prototype for alignment with the ISO/IEC 9899:1999 standard.

References to the `wait3()` function are removed.

The SYNOPSIS is marked CX since the presence of this function in the `<signal.h>` header is an extension over the ISO C standard.
NAME
sigaddset — add a signal to a signal set

SYNOPSIS
CX
#include <signal.h>

int sigaddset(sigset_t *set, int signo);

DESCRIPTION
The sigaddset() function adds the individual signal specified by the signo
to the signal set pointed to by set.

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.

RETURN VALUE
Upon successful completion, sigaddset() shall return 0; otherwise, it shall return −1 and set errno
to indicate the error.

ERRORS
The sigaddset() function may fail if:

[EINVAL] The value of the signo argument is an invalid or unsupported signal number.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.4 (on page 28), sigaction(), sigdelset(), sigemptyset(), sigfillset(), sigismember(),
sigpending(), sigprocmask(), sigsuspend(), the Base Definitions volume of IEEE Std 1003.1-2001,
<signal.h>

CHANGE HISTORY
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 5
The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in
previous issues.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an
extension over the ISO C standard.
**NAME**
sigaltstack — set and get signal alternate stack context

**SYNOPSIS**
```c
#include <signal.h>

int sigaltstack(const stack_t *restrict ss, stack_t *restrict oss);
```

**DESCRIPTION**
The `sigaltstack()` function allows a process to define and examine the state of an alternate stack for signal handlers. Signals that have been explicitly declared to execute on the alternate stack shall be delivered on the alternate stack.

If `ss` is not a null pointer, it points to a `stack_t` structure that specifies the alternate signal stack that shall take effect upon return from `sigaltstack()`. The `ss_flags` member specifies the new stack state. If it is set to `SS_DISABLE`, the stack is disabled and `ss_sp` and `ss_size` are ignored. Otherwise, the stack shall be enabled, and the `ss_sp` and `ss_size` members specify the new address and size of the stack.

The range of addresses starting at `ss_sp` up to but not including `ss_sp+ss_size` is available to the implementation for use as the stack. This function makes no assumptions regarding which end is the stack base and in which direction the stack grows as items are pushed.

If `oss` is not a null pointer, on successful completion it shall point to a `stack_t` structure that specifies the alternate signal stack that was in effect prior to the call to `sigaltstack()`. The `ss_sp` and `ss_size` members specify the address and size of that stack. The `ss_flags` member specifies the stack's state, and may contain one of the following values:

- **SS_ONSTACK** The process is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the process is executing on it fail. This flag shall not be modified by processes.
- **SS_DISABLE** The alternate signal stack is currently disabled.

The value `SIGSTKSZ` is a system default specifying the number of bytes that would be used to cover the usual case when manually allocating an alternate stack area. The value `MINSIGSTKSZ` is defined to be the minimum stack size for a signal handler. In computing an alternate stack size, a program should add that amount to its stack requirements to allow for the system implementation overhead. The constants `SS_ONSTACK`, `SS_DISABLE`, `SIGSTKSZ`, and `MINSIGSTKSZ` are defined in `<signal.h>`.

After a successful call to one of the `exec` functions, there are no alternate signal stacks in the new process image.

In some implementations, a signal (whether or not indicated to execute on the alternate stack) shall always execute on the alternate stack if it is delivered while another signal is being caught using the alternate stack.

Use of this function by library threads that are not bound to kernel-scheduled entities results in undefined behavior.

**RETURN VALUE**
Upon successful completion, `sigaltstack()` shall return 0; otherwise, it shall return −1 and set `errno` to indicate the error.
ERRORS

The `sigaltstack()` function shall fail if:

- **[EINVAL]** The `ss` argument is not a null pointer, and the `ss_flags` member pointed to by `ss` contains flags other than `SS_DISABLE`.
- **[ENOMEM]** The size of the alternate stack area is less than `MINSIGSTKSZ`.
- **[EPERM]** An attempt was made to modify an active stack.

EXAMPLES

Allocating Memory for an Alternate Stack

The following example illustrates a method for allocating memory for an alternate stack.

```c
#include <signal.h>
...
if ((sigstk.ss_sp = malloc(SIGSTKSZ)) == NULL)
   /* Error return. */
   sigstk.ss_size = SIGSTKSZ;
   sigstk.ss_flags = 0;
   if (sigaltstack(&sigstk, (stack_t *)0) < 0)
      perror("sigaltstack");
```

APPLICATION USAGE

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 28), `sigaction()`, `sigsetjmp()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`

CHANGE HISTORY

First released in Issue 4, Version 2.

**Issue 5**

Moved from X/OPEN UNIX extension to BASE.

The last sentence of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

**Issue 6**

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The `restrict` keyword is added to the `sigaltstack()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME

sigdelset — delete a signal from a signal set

SYNOPSIS

```c
#include <signal.h>

int sigdelset(sigset_t *set, int signo);
```

DESCRIPTION

The `sigdelset()` function deletes the individual signal specified by `signo` from the signal set pointed to by `set`.

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.

RETURN VALUE

Upon successful completion, `sigdelset()` shall return 0; otherwise, it shall return −1 and set `errno` to indicate the error.

ERRORS

The `sigdelset()` function may fail if:

- `[EINVAL]` The `signo` argument is not a valid signal number, or is an unsupported signal number.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

- Section 2.4 (on page 28), `sigaction()`, `sigaddset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `sigpending()`, `sigprocmask()`, `sigsuspend()`, `sigtimedwait()`, `sigwait()`, or `sigwaitinfo()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

Issue 6

The SYNOPSIS is marked CX since the presence of this function in the `<signal.h>` header is an extension over the ISO C standard.
**NAME**  
sigemptyset — initialize and empty a signal set

**SYNOPSIS**

```c
#include <signal.h>

int sigemptyset(sigset_t *set);
```

**DESCRIPTION**

The `sigemptyset()` function initializes the signal set pointed to by `set`, such that all signals defined in IEEE Std 1003.1-2001 are excluded.

**RETURN VALUE**

Upon successful completion, `sigemptyset()` shall return 0; otherwise, it shall return −1 and set `errno` to indicate the error.

**ERRORS**

No errors are defined.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

The implementation of the `sigemptyset()` (or `sigfillset()`) function could quite trivially clear (or set) all the bits in the signal set. Alternatively, it would be reasonable to initialize part of the structure, such as a version field, to permit binary-compatibility between releases where the size of the set varies. For such reasons, either `sigemptyset()` or `sigfillset()` must be called prior to any other use of the signal set, even if such use is read-only (for example, as an argument to `sigpending()`). This function is not intended for dynamic allocation.

The `sigfillset()` and `sigemptyset()` functions require that the resulting signal set include (or exclude) all the signals defined in this volume of IEEE Std 1003.1-2001. Although it is outside the scope of this volume of IEEE Std 1003.1-2001 to place this requirement on signals that are implemented as extensions, it is recommended that implementation-defined signals also be affected by these functions. However, there may be a good reason for a particular signal not to be affected. For example, blocking or ignoring an implementation-defined signal may have undesirable side effects, whereas the default action for that signal is harmless. In such a case, it would be preferable for such a signal to be excluded from the signal set returned by `sigfillset()`.

In early proposals there was no distinction between invalid and unsupported signals (the names of optional signals that were not supported by an implementation were not defined by that implementation). The [EINVAL] error was thus specified as a required error for invalid signals. With that distinction, it is not necessary to require implementations of these functions to determine whether an optional signal is actually supported, as that could have a significant performance impact for little value. The error could have been required for invalid signals and optional for unsupported signals, but this seemed unnecessarily complex. Thus, the error is optional in both cases.

**FUTURE DIRECTIONS**

None.
SEE ALSO

Section 2.4 (on page 28), `sigaction()`, `sigaddset()`, `sigdelset()`, `sigfillset()`, `sigismember()`, `sigpending()`, `sigprocmask()`, `sigsuspend()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 6

The SYNOPSIS is marked CX since the presence of this function in the `<signal.h>` header is an extension over the ISO C standard.
sigfillset()

NAME
    sigfillset — initialize and fill a signal set

SYNOPSIS
    CX
    #include <signal.h>
    int sigfillset(sigset_t *set);

DESCRIPTION
    The sigfillset() function shall initialize the signal set pointed to by
    set, such that all signals defined in this volume of IEEE Std 1003.1-2001 are included.

RETURN VALUE
    Upon successful completion, sigfillset() shall return 0; otherwise, it shall return −1 and set errno
    to indicate the error.

ERRORS
    No errors are defined.

EXAMPLES
    None.

APPLICATION USAGE
    None.

RATIONALE
    Refer to sigemptyset() (on page 1348).

FUTURE DIRECTIONS
    None.

SEE ALSO
    Section 2.4 (on page 28), sigaction(), sigaddset(), sigdelset(), sigemptyset(), sigismember(),
    sigpending(), sigprocmask(), sigsuspend(), the Base Definitions volume of IEEE Std 1003.1-2001,
    <signal.h>

CHANGE HISTORY
    First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 6
    The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an
    extension over the ISO C standard.
NAME
sighold(), sigignore, sigpause, sigrelse, sigset — signal management

SYNOPSIS
#include <signal.h>

int sighold(int sig);
int sigignore(int sig);
int sigpause(int sig);
int sigrelse(int sig);
void (*sigset(int sig, void (*disp)(int)))(int);

DESCRIPTION
Use of any of these functions is unspecified in a multi-threaded process.

The sighold(), sigignore(), sigpause(), sigrelse(), and sigset() functions provide simplified signal management.

The sigset() function shall modify signal dispositions. The sig argument specifies the signal, which may be any signal except SIGKILL and SIGSTOP. The disp argument specifies the signal’s disposition, which may be SIG_DFL, SIG_IGN, or the address of a signal handler. If sigset() is used, and disp is the address of a signal handler, the system shall add sig to the calling process’ signal mask before executing the signal handler; when the signal handler returns, the system shall restore the calling process’ signal mask to its state prior to the delivery of the signal. In addition, if sigset() is used, and disp is equal to SIG_HOLD, sig shall be added to the calling process’ signal mask and sig’s disposition shall remain unchanged. If sigset() is used, and disp is not equal to SIG_HOLD, sig shall be removed from the calling process’ signal mask.

The sighold() function shall add sig to the calling process’ signal mask.

The sigrelse() function shall remove sig from the calling process’ signal mask.

The sigignore() function shall set the disposition of sig to SIG_IGN.

The sigpause() function shall remove sig from the calling process’ signal mask and suspend the calling process until a signal is received. The sigpause() function shall restore the process’ signal mask to its original state before returning.

If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the calling processes shall not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it shall block until all of its children terminate, and wait(), waitid(), and waitpid() shall fail and set errno to [ECHILD].

RETURN VALUE
Upon successful completion, sigset() shall return SIG_HOLD if the signal had been blocked and the signal’s previous disposition if it had not been blocked. Otherwise, SIG_ERR shall be returned and errno set to indicate the error.

The sigpause() function shall suspend execution of the thread until a signal is received, whereupon it shall return −1 and set errno to [EINTR].

For all other functions, upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to indicate the error.
ERRORS

These functions shall fail if:

[EINVAL] The sig argument is an illegal signal number.

The sigset() and sigignore() functions shall fail if:

[EINVAL] An attempt is made to catch a signal that cannot be caught, or to ignore a
signal that cannot be ignored.

EXAMPLES

None.

APPLICATION USAGE

The sigaction() function provides a more comprehensive and reliable mechanism for controlling
signals; new applications should use sigaction() rather than sigset().

The sighold() function, in conjunction with sigrelse() or sigpause(), may be used to establish
critical regions of code that require the delivery of a signal to be temporarily deferred.

The sigsuspend() function should be used in preference to sigpause() for broader portability.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 28), exec, pause(), sigaction(), signal(), sigsuspend(), waitid(), the Base
Definitions volume of IEEE Std 1003.1-2001, <signal.h>

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated to indicate that the sigpause() function restores the process’
signal mask to its original state before returning.

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].

Issue 6

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

References to the wait3() function are removed.

The XSI functions are split out into their own reference page.
NAME
 siginterrupt — allow signals to interrupt functions

SYNOPSIS
 XSI
 #include <signal.h>

 int siginterrupt(int sig, int flag);

DESCRIPTION
 The siginterrupt() function shall change the restart behavior when a function is interrupted by
 the specified signal. The function siginterrupt(sig, flag) has an effect as if implemented as:

 siginterrupt(int sig, int flag) {
 int ret;
 struct sigaction act;
 (void) sigaction(sig, NULL, &act);
 if (flag)
   act.sa_flags &= ~SA_RESTART;
 else
   act.sa_flags |= SA_RESTART;
 ret = sigaction(sig, &act, NULL);
 return ret;
 }

RETURN VALUE
 Upon successful completion, siginterrupt() shall return 0; otherwise, −1 shall be returned and
 errno set to indicate the error.

ERRORS
 The siginterrupt() function shall fail if:

 [EINVAL] The sig argument is not a valid signal number.

EXAMPLES
 None.

APPLICATION USAGE
 The siginterrupt() function supports programs written to historical system interfaces. A
 conforming application, when being written or rewritten, should use sigaction() with the
 SA_RESTART flag instead of siginterrupt().

RATIONALE
 None.

FUTURE DIRECTIONS
 None.

SEE ALSO
 Section 2.4 (on page 28), sigaction(), the Base Definitions volume of IEEE Std 1003.1-2001,
 <signal.h>

CHANGE HISTORY
 First released in Issue 4, Version 2.
Moved from X/OPEN UNIX extension to BASE.
NAME
sigismember — test for a signal in a signal set

SYNOPSIS
#include <signal.h>

int sigismember(const sigset_t *set, int signo);

DESCRIPTION
The sigismember() function shall test whether the signal specified by signo is a member of the set pointed to by set.

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.

RETURN VALUE
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.

ERRORS
The sigismember() function may fail if:

[EINVAL] The signo argument is not a valid signal number, or is an unsupported signal number.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.4 (on page 28), sigaction(), sigaddset(), sigdelset(), sigfillset(), sigemptyset(), sigpending(), sigprocmask(), sigsuspend(), the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>

CHANGE HISTORY
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 5
The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

Issue 6
The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an extension over the ISO C standard.
NAME

siglongjmp — non-local goto with signal handling

SYNOPSIS

```c
#include <setjmp.h>

void siglongjmp(sigjmp_buf env, int val);
```

DESCRIPTION

The `siglongjmp()` function shall be equivalent to the `longjmp()` function, except as follows:

- References to `setjmp()` shall be equivalent to `sigsetjmp()`.
- The `siglongjmp()` function shall restore the saved signal mask if and only if the `env` argument was initialized by a call to `sigset jmp()` with a non-zero `savemask` argument.

RETURN VALUE

After `siglongjmp()` is completed, program execution shall continue as if the corresponding invocation of `sigset jmp()` had just returned the value specified by `val`. The `siglongjmp()` function shall not cause `sigset jmp()` to return 0; if `val` is 0, `sigsetjmp()` shall return the value 1.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The distinction between `set jmp()` or `long jmp()` and `sigset jmp()` or `siglong jmp()` is only significant for programs which use `sigaction()`, `sig procmask()`, or `sigsuspend()`.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`long jmp()`, `set jmp()`, `sig procmask()`, `sigset jmp()`, `sigsuspend()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<set jmp.h>`

CHANGE HISTORY

First released in Issue 3. Included for alignment with the ISO POSIX-1 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The DESCRIPTION is rewritten in terms of `long jmp()`.

The SYNOPSIS is marked CX since the presence of this function in the `<set jmp.h>` header is an extension over the ISO C standard.
NAME
signal — signal management

SYNOPSIS
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);

DESCRIPTION
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.

Use of this function is unspecified in a multi-threaded process.

The signal() function chooses one of three ways in which receipt of the signal number sig is to be subsequently handled. If the value of func is SIG_DFL, default handling for that signal shall occur. If the value of func is SIG_IGN, the signal shall be ignored. Otherwise, the application shall ensure that func points to a function to be called when that signal occurs. An invocation of such a function because of a signal, or (recursively) of any further functions called by that invocation (other than functions in the standard library), is called a “signal handler”.

When a signal occurs, and func points to a function, it is implementation-defined whether the equivalent of:

signal(sig, SIG_DFL);

is executed or the implementation prevents some implementation-defined set of signals (at least including sig) from occurring until the current signal handling has completed. (If the value of sig is SIGILL, the implementation may alternatively define that no action is taken.) Next the equivalent of:

(*func)(sig);

is executed. If and when the function returns, if the value of sig was SIGFPE, SIGILL, or SIGSEGV or any other implementation-defined value corresponding to a computational exception, the behavior is undefined. Otherwise, the program shall resume execution at the point it was interrupted. If the signal occurs as the result of calling the abort(), raise(), kill(), pthread_kill(), or sigqueue() function, the signal handler shall not call the raise() function.

If the signal occurs other than as the result of calling abort(), raise(), kill(), pthread_kill(), or sigqueue(), the behavior is undefined if the signal handler refers to any object with static storage duration other than by assigning a value to an object declared as volatile sig_atomic_t, or if the signal handler calls any function in the standard library other than one of the functions listed in Section 2.4 (on page 28). Furthermore, if such a call fails, the value of errno is unspecified.

At program start-up, the equivalent of:

signal(sig, SIG_IGN);

is executed for some signals, and the equivalent of:

signal(sig, SIG_DFL);

is executed for all other signals (see exec).

RETURN VALUE
If the request can be honored, signal() shall return the value of func for the most recent call to signal() for the specified signal sig. Otherwise, SIG_ERR shall be returned and a positive value shall be stored in errno.
The \texttt{signal()} function shall fail if:

\begin{verbatim}
CX [EINVAL] The \texttt{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.
\end{verbatim}

The \texttt{signal()} function may fail if:

\begin{verbatim}
CX [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be caught or ignored (or both).
\end{verbatim}

\textbf{EXAMPLES}

None.

\textbf{APPLICATION USAGE}

The \texttt{sigaction()} function provides a more comprehensive and reliable mechanism for controlling signals; new applications should use \texttt{sigaction()} rather than \texttt{signal()}.

\textbf{RATIONALE}

None.

\textbf{FUTURE DIRECTIONS}

None.

\textbf{SEE ALSO}

Section 2.4 (on page 28), \texttt{exec}, \texttt{pause()}, \texttt{sigaction()}, \texttt{sigsuspend()}, \texttt{waitid()}, the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<signal.h>}

\textbf{CHANGE HISTORY}

First released in Issue 1. Derived from Issue 1 of the SVID.

\textbf{Issue 5}

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated to indicate that the \texttt{sigpause()} function restores the process’ signal mask to its original state before returning.

The RETURN VALUE section is updated to indicate that the \texttt{sigpause()} function suspends execution of the process until a signal is received, whereupon it returns \texttt{−1} and sets \texttt{errno} to [EINTR].

\textbf{Issue 6}

Extensions beyond the ISO C standard are marked.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

References to the \texttt{wait3()} function are removed.

The \texttt{sighold()}, \texttt{sigignore()}, \texttt{sigrelse()}, and \texttt{sigset()} functions are split out onto their own reference page.
NAME
signbit — test sign

SYNOPSIS
#include <math.h>
int signbit(real-floating x);

DESCRIPTION
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 signbit() macro shall determine whether the sign of its argument value is negative. NaNs, zeros, and infinities have a sign bit.

RETURN VALUE
The signbit() macro shall return a non-zero value if and only if the sign of its argument value is negative.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fpclassify(), finite(), isinf(), isnan(), isnormal(), the Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

CHANGE HISTORY
NAME

sigpause — remove a signal from the signal mask and suspend the thread

SYNOPSIS

XSI

#include <signal.h>

int sigpause(int sig);

DESCRIPTION

Refer to sighold().
NAME
sigpending — examine pending signals

SYNOPSIS
CX
#include <signal.h>

int sigpending(sigset_t *set);

DESCRIPTION
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.

RETURN VALUE
Upon successful completion, sigpending() shall return 0; otherwise, −1 shall be returned and
errno set to indicate the error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
sigaddset(), sigdelset(), sigemptyset(), sigfillset(), sigismember(), sigprocmask(), the Base Definitions
volume of IEEE Std 1003.1-2001, <signal.h>

CHANGE HISTORY
First released in Issue 3.

Issue 5
The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6
The SYNOPSIS is marked CX since the presence of this function in the <signal.h> header is an
extension over the ISO C standard.
NAME
sigprocmask — examine and change blocked signals

SYNOPSIS
```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
                 sigset_t *restrict oset);
```

DESCRIPTION
Refer to pthread_sigmask().
NAME

sigqueue — queue a signal to a process (REALTIME)

SYNOPSIS

#include <signal.h>

int sigqueue(pid_t pid, int signo, const union sigval value);

DESCRIPTION

The sigqueue() function shall cause the signal specified by signo to be sent with the value specified by value to the process specified by pid. If signo is zero (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of pid.

The conditions required for a process to have permission to queue a signal to another process are the same as for the kill() function.

The sigqueue() function shall return immediately. If SA_SIGINFO is set for signo and if the resources were available to queue the signal, the signal shall be queued and sent to the receiving process. If SA_SIGINFO is not set for signo, then signo shall be sent at least once to the receiving process; it is unspecified whether value shall be sent to the receiving process as a result of this call.

If the value of pid causes signo to be generated for the sending process, and if signo is not blocked for the calling thread and if no other thread has signo unblocked or is waiting in a sigwait() function for signo, either signo or at least the pending, unblocked signal shall be delivered to the calling thread before the sigqueue() function returns. Should any multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected for delivery, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified.

RETURN VALUE

Upon successful completion, the specified signal shall have been queued, and the sigqueue() function shall return a value of zero. Otherwise, the function shall return a value of -1 and set errno to indicate the error.

ERRORS

The sigqueue() function shall fail if:

[EAGAIN] No resources are available to queue the signal. The process has already queued [SIGQUEUE_MAX] signals that are still pending at the receiver(s), or a system-wide resource limit has been exceeded.

[EINVAL] The value of the signo argument is an invalid or unsupported signal number.

[EPERM] The process does not have the appropriate privilege to send the signal to the receiving process.

[ESRCH] The process pid does not exist.
The `sigqueue()` function allows an application to queue a realtime signal to itself or to another process, specifying the application-defined value. This is common practice in realtime applications on existing realtime systems. It was felt that specifying another function in the `sig...` name space already carved out for signals was preferable to extending the interface to `kill()`. Such a function became necessary when the put/get event function of the message queues was removed. It should be noted that the `sigqueue()` function implies reduced performance in a security-conscious implementation as the access permissions between the sender and receiver have to be checked on each send when the `pid` is resolved into a target process. Such access checks were necessary only at message queue open in the previous interface.

The standard developers required that `sigqueue()` have the same semantics with respect to the null signal as `kill()`, and that the same permission checking be used. But because of the difficulty of implementing the “broadcast” semantic of `kill()` (for example, to process groups) and the interaction with resource allocation, this semantic was not adopted. The `sigqueue()` function queues a signal to a single process specified by the `pid` argument.

The `sigqueue()` function can fail if the system has insufficient resources to queue the signal. An explicit limit on the number of queued signals that a process could send was introduced. While the limit is “per-sender”, this volume of IEEE Std 1003.1-2001 does not specify that the resources be part of the state of the sender. This would require either that the sender be maintained after exit until all signals that it had sent to other processes were handled or that all such signals that had not yet been acted upon be removed from the queue(s) of the receivers. This volume of IEEE Std 1003.1-2001 does not preclude this behavior, but an implementation that allocated queuing resources from a system-wide pool (with per-sender limits) and that leaves queued signals pending after the sender exits is also permitted.

The `sigqueue()` function is marked as part of the Realtime Signals Extension option. The `[ENOSYS]` error condition has been removed as stubs need not be provided if an implementation does not support the Realtime Signals Extension option.
NAME
sigrelse, sigset — signal management

SYNOPSIS
XSI

```c
#include <signal.h>

int sigrelse(int sig);
void (*sigset(int sig, void (*disp)(int)))(int);
```

DESCRIPTION
Refer to `sighold()`.
sigsetjmp() — set jump point for a non-local goto

The `sigsetjmp()` function shall be equivalent to the `setjmp()` function, except as follows:

- References to `setjmp()` are equivalent to `sigsetjmp()`.
- References to `longjmp()` are equivalent to `siglongjmp()`.
- If the value of the `savemask` argument is not 0, `sigsetjmp()` shall also save the current signal mask of the calling thread as part of the calling environment.

If the return is from a successful direct invocation, `sigsetjmp()` shall return 0. If the return is from a call to `siglongjmp()`, `sigsetjmp()` shall return a non-zero value.

No errors are defined.

The distinction between `setjmp()`/`longjmp()` and `sigsetjmp()`/`siglongjmp()` is only significant for programs which use `sigaction()`, `sigprocmask()`, or `sigsuspend()`.

Note that since this function is defined in terms of `setjmp()`, if `savemask` is zero, it is unspecified whether the signal mask is saved.

The ISO C standard specifies various restrictions on the usage of the `setjmp()` macro in order to permit implementors to recognize the name in the compiler and not implement an actual function. These same restrictions apply to the `sigsetjmp()` macro.

There are processors that cannot easily support these calls, but this was not considered a sufficient reason to exclude them.

4.2 BSD, 4.3 BSD, and XSI-conformant systems provide functions named `_setjmp()` and `_longjmp()` that, together with `setjmp()` and `longjmp()`, provide the same functionality as `sigsetjmp()` and `siglongjmp()`. On those systems, `setjmp()` and `longjmp()` save and restore signal masks, while `_setjmp()` and `_longjmp()` do not. On System V Release 3 and in corresponding issues of the SVID, `setjmp()` and `longjmp()` are explicitly defined not to save and restore signal masks. In order to permit existing practice in both cases, the relation of `setjmp()` and `longjmp()` to signal masks is not specified, and a new set of functions is defined instead.

The `longjmp()` and `siglongjmp()` functions operate as in the previous issue provided the matching `setjmp()` or `sigsetjmp()` has been performed in the same thread. Non-local jumps into contexts saved by other threads would be at best a questionable practice and were not considered worthy of standardization.
FUTURE DIRECTIONS
None.

SEE ALSO
siglongjmp(), signal(), sigprocmask(), sigsuspend(), the Base Definitions volume of
IEEE Std 1003.1-2001, <setjmp.h>

CHANGE HISTORY
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 5
The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6
The DESCRIPTION is reworded in terms of setjmp().
The SYNOPTIS is marked CX since the presence of this function in the <setjmp.h> header is an
extension over the ISO C standard.
sigsuspend() — wait for a signal

#include <signal.h>

int sigsuspend(const sigset_t *sigmask);

The `sigsuspend()` function shall replace the current signal mask of the calling thread with the set of signals pointed to by `sigmask` and then suspend the thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. This shall not cause any other signals that may have been pending on the process to become pending on the thread.

If the action is to terminate the process then `sigsuspend()` shall never return. If the action is to execute a signal-catching function, then `sigsuspend()` shall return after the signal-catching function returns, with the signal mask restored to the set that existed prior to the `sigsuspend()` call.

It is not possible to block signals that cannot be ignored. This is enforced by the system without causing an error to be indicated.

Since `sigsuspend()` suspends thread execution indefinitely, there is no successful completion return value. If a return occurs, −1 shall be returned and `errno` set to indicate the error.

The `sigsuspend()` function shall fail if:

A signal is caught by the calling process and control is returned from the signal-catching function.

Normally, at the beginning of a critical code section, a specified set of signals is blocked using the `sigprocmask()` function. When the thread has completed the critical section and needs to wait for the previously blocked signal(s), it pauses by calling `sigsuspend()` with the mask that was returned by the `sigprocmask()` call.

None.

None.

None.

Section 2.4 (on page 28), `pause()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.
42465 **Issue 5**
42466 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

42467 **Issue 6**
42468 The text in the RETURN VALUE section has been changed from “suspends process execution” to “suspends thread execution”. This reflects IEEE PASC Interpretation 1003.1c #40.
42469 Text in the APPLICATION USAGE section has been replaced.
42470 The SYNOPSIS is marked CX since the presence of this function in the `<signal.h>` header is an extension over the ISO C standard.
NAME
sigtimedwait, sigwaitinfo — wait for queued signals (REALTIME)

SYNOPSIS
#include <signal.h>

int sigtimedwait(const sigset_t *restrict set,
                  siginfo_t *restrict info,
                  const struct timespec *restrict timeout);

int sigwaitinfo(const sigset_t *restrict set,
                 siginfo_t *restrict info);

DESCRIPTION
The sigtimedwait() function shall be equivalent to sigwaitinfo() except that if none of the signals
specified by set are pending, sigtimedwait() shall wait for the time interval specified in the
timespec structure referenced by timeout. If the timespec structure pointed to by timeout is
zero-valued and if none of the signals specified by set are pending, then sigtimedwait() shall
return immediately with an error. If timeout is the NULL pointer, the behavior is unspecified. If
the Monotonic Clock option is supported, the CLOCK_MONOTONIC clock shall be used to
measure the time interval specified by the timeout argument.

The sigwaitinfo() function selects the pending signal from the set specified by set. Should any of
multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it shall be the
lowest numbered one. The selection order between realtime and non-realtime signals, or
between multiple pending non-realtime signals, is unspecified. If no signal in set is pending at
the time of the call, the calling thread shall be suspended until one or more signals in set become
pending or until it is interrupted by an unblocked, caught signal.

The sigwaitinfo() function shall be equivalent to the sigwait() function if the info argument is
NULL. If the info argument is non-NULL, the sigwaitinfo() function shall be equivalent to
sigwait(), except that the selected signal number shall be stored in the si_signo member, and the
cause of the signal shall be stored in the si_code member. If any value is queued to the selected
signal, the first such queued value shall be dequeued and, if the info argument is non-NULL, the
value shall be stored in the si_value member of info. The system resource used to queue the
signal shall be released and returned to the system for other use. If no value is queued, the
content of the si_value member is undefined. If no further signals are queued for the selected
signal, the pending indication for that signal shall be reset.

RETURN VALUE
Upon successful completion (that is, one of the signals specified by set is pending or is
generated) sigwaitinfo() and sigtimedwait() shall return the selected signal number. Otherwise,
the function shall return a value of −1 and set errno to indicate the error.

ERRORS
The sigtimedwait() function shall fail if:

[EAGAIN] No signal specified by set was generated within the specified timeout period.

The sigtimedwait() and sigwaitinfo() functions may fail if:

[EINTR] The wait was interrupted by an unblocked, caught signal. It shall be
documented in system documentation whether this error causes these
functions to fail.

The sigtimedwait() function may also fail if:
The `sigtimedwait()` function times out and returns an [EINVAL] error. Application writers should note that this is inconsistent with other functions such as `pthread_cond_timedwait()` that return [ETIMEDOUT].

Existing programming practice on real-time systems uses the ability to pause waiting for a selected set of events and handle the first event that occurs in-line instead of in a signal-handling function. This allows applications to be written in an event-directed style similar to a state machine. This style of programming is useful for large-scale transaction processing in which the overall throughput of an application and the ability to clearly track states are more important than the ability to minimize the response time of individual event handling.

It is possible to construct a signal-waiting macro function out of the real-time signal function mechanism defined in this volume of IEEE Std 1003.1-2001. However, such a macro has to include the definition of a generalized handler for all signals to be waited on. A significant portion of the overhead of handler processing can be avoided if the signal-waiting function is provided by the kernel. This volume of IEEE Std 1003.1-2001 therefore provides two signal-waiting functions—one that waits indefinitely and one with a timeout—as part of the overall real-time signal function specification.

The specification of a function with a timeout allows an application to be written that can be broken out of a wait after a set period of time if no event has occurred. It was argued that setting a timer event before the wait and recognizing the timer event in the wait would also implement the same functionality, but at a lower performance level. Because of the performance degradation associated with the user-level specification of a timer event and the subsequent cancelation of that timer event after the wait completes for a valid event, and the complexity associated with handling potential race conditions associated with the user-level method, the separate function has been included.

Note that the semantics of the `sigwaitinfo()` function are nearly identical to that of the `sigwait()` function defined by this volume of IEEE Std 1003.1-2001. The only difference is that `sigwaitinfo()` returns the queued signal value in the `value` argument. The return of the queued value is required so that applications can differentiate between multiple events queued to the same signal number.

The two distinct functions are being maintained because some implementations may choose to implement the POSIX Threads Extension functions and not implement the queued signals extensions. Note, though, that `sigwaitinfo()` does not return the queued value if the `value` argument is NULL, so the POSIX Threads Extension `sigwait()` function can be implemented as a macro on `sigwaitinfo()`.

The `sigtimedwait()` function was separated from the `sigwaitinfo()` function to address concerns regarding the overloading of the `timeout` pointer to indicate indefinite wait (no timeout), timed wait, and immediate return, and concerns regarding consistency with other functions where the conditional and timed waits were separate functions from the pure blocking function. The semantics of `sigtimedwait()` are specified such that `sigwaitinfo()` could be implemented as a macro with a NULL pointer for `timeout`. 

The `timeout` argument specified a `tv_nsec` value less than zero or greater than or equal to 1 000 million.
The `sigwait` functions provide a synchronous mechanism for threads to wait for asynchronously-generated signals. One important question was how many threads that are suspended in a call to a `sigwait()` function for a signal should return from the call when the signal is sent. Four choices were considered:

1. Return an error for multiple simultaneous calls to `sigwait` functions for the same signal.
2. One or more threads return.
3. All waiting threads return.
4. Exactly one thread returns.

Prohibiting multiple calls to `sigwait()` for the same signal was felt to be overly restrictive. The “one or more” behavior made implementation of conforming packages easy at the expense of forcing POSIX threads clients to protect against multiple simultaneous calls to `sigwait()` in application code in order to achieve predictable behavior. There was concern that the “all waiting threads” behavior would result in “signal broadcast storms”, consuming excessive CPU resources by replicating the signals in the general case. Furthermore, no convincing examples could be presented that delivery to all was either simpler or more powerful than delivery to one.

Thus, the consensus was that exactly one thread that was suspended in a call to a `sigwait` function for a signal should return when that signal occurs. This is not an onerous restriction as:

- A multi-way signal wait can be built from the single-way wait.
- Signals should only be handled by application-level code, as library routines cannot guess what the application wants to do with signals generated for the entire process.
- Applications can thus arrange for a single thread to wait for any given signal and call any needed routines upon its arrival.

In an application that is using signals for interprocess communication, signal processing is typically done in one place. Alternatively, if the signal is being caught so that process cleanup can be done, the signal handler thread can call separate process cleanup routines for each portion of the application. Since the application main line started each portion of the application, it is at the right abstraction level to tell each portion of the application to clean up.

Certainly, there exist programming styles where it is logical to consider waiting for a single signal in multiple threads. A simple `sigwait_multiple()` routine can be constructed to achieve this goal. A possible implementation would be to have each `sigwait_multiple()` caller registered as having expressed interest in a set of signals. The caller then waits on a thread-specific condition variable. A single server thread calls a `sigwait()` function on the union of all registered signals. When the `sigwait()` function returns, the appropriate state is set and condition variables are broadcast. New `sigwait_multiple()` callers may cause the pending `sigwait()` call to be canceled and reissued in order to update the set of signals being waited for.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

Section 2.8.1 (on page 41), `pause()`, `pthread_sigmask()`, `sigaction()`, `sigpending()`, `sigsuspend()`, `sigwait()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<signal.h>`, `<time.h>`

**CHANGE HISTORY**

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.
These functions are marked as part of the Realtime Signals Extension option.

The Open Group Corrigendum U035/3 is applied. The SYNOPSIS of the `sigwaitinfo()` function has been corrected so that the second argument is of type `siginfo_t *`.

The `ENOSYS` error condition has been removed as stubs need not be provided if an implementation does not support the Realtime Signals Extension option.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the CLOCK_MONOTONIC clock, if supported, is used to measure timeout intervals.

The `restrict` keyword is added to the `sigtimedwait()` and `sigwaitinfo()` prototypes for alignment with the ISO/IEC 9899:1999 standard.
NAME
sigwait — wait for queued signals

SYNOPSIS
CX
#include <signal.h>

int sigwait(const sigset_t *restrict set, int *restrict sig);

DESCRIPTION
The sigwait() function shall select a pending signal from set, atomically clear it from the system's set of pending signals, and return that signal number in the location referenced by sig. If prior to the call to sigwait() there are multiple pending instances of a single signal number, it is implementation-defined whether upon successful return there are any remaining pending signals for that signal number. If the implementation supports queued signals and there are multiple signals queued for the signal number selected, the first such queued signal shall cause a return from sigwait() and the remainder shall remain queued. If no signal in set is pending at the time of the call, the thread shall be suspended until one or more becomes pending. The signals defined by set shall have been blocked at the time of the call to sigwait(); otherwise, the behavior is undefined. The effect of sigwait() on the signal actions for the signals in set is unspecified.

If more than one thread is using sigwait() to wait for the same signal, no more than one of these threads shall return from sigwait() with the signal number. Which thread returns from sigwait() if more than a single thread is waiting is unspecified.

RTS Should any of the multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified.

RETURN VALUE
Upon successful completion, sigwait() shall store the signal number of the received signal at the location referenced by sig and return zero. Otherwise, an error number shall be returned to indicate the error.

ERRORS
The sigwait() function may fail if:

[EINVAL] The set argument contains an invalid or unsupported signal number.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
To provide a convenient way for a thread to wait for a signal, this volume of IEEE Std 1003.1-2001 provides the sigwait() function. For most cases where a thread has to wait for a signal, the sigwait() function should be quite convenient, efficient, and adequate.

However, requests were made for a lower-level primitive than sigwait() and for semaphores that could be used by threads. After some consideration, threads were allowed to use semaphores and sem_post() was defined to be async-signal and async-cancel-safe.

In summary, when it is necessary for code run in response to an asynchronous signal to notify a thread, sigwait() should be used to handle the signal. Alternatively, if the implementation provides semaphores, they also can be used, either following sigwait() or from within a signal handling routine previously registered with sigaction().
FUTURE DIRECTIONS

None.

SEE ALSO
Section 2.4 (on page 28), Section 2.8.1 (on page 41), pause(), pthread_sigmask(), sigaction(), sigpending(), sigsuspend(), sigwaitinfo(), the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>, <time.h>

CHANGE HISTORY
First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6
The restrict keyword is added to the sigwait() prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
   sigwaitinfo — wait for queued signals (REALTIME)

SYNOPSIS
   #include <signal.h>

   int sigwaitinfo(const sigset_t *restrict set, siginfo_t *restrict info);

DESCRIPTION
   Refer to sigtimedwait().
NAME
sin, sinf, sinl — sine function

SYNOPSIS
#include <math.h>

double sin(double x);
float sinf(float x);
long double sinl(long double x);

DESCRIPTION
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
These functions shall compute the sine of their argument x, measured in radians.
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.

RETURN VALUE
Upon successful completion, these functions shall return the sine of x.

MX If x is NaN, a NaN shall be returned.
If x is ±0, x shall be returned.
If x is subnormal, a range error may occur and x should be returned.
If x is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS
These functions shall fail if:

MX Domain Error The x argument is ±Inf.
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.

These functions may fail if:

MX Range Error The value of x is subnormal
If the integer expression (math_errhandling & MATH_ERRNO) is non-zero,
then errno shall be set to [ERANGE]. If the integer expression
(mmath_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow
floating-point exception shall be raised.
EXAMPLES

Taking the Sine of a 45-Degree Angle

```c
#include <math.h>
...
double radians = 45.0 * M_PI / 180;
double result;
...
result = sin(radians);
```

APPLICATION USAGE

These functions may lose accuracy when their argument is near a multiple of \( \pi \) or is far from 0.0.

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`asinf()`, `fecl except()`, `fetestexcept()`, `isnan()`, the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, `<math.h>`

CHANGE HISTORY

Issue 5

The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

Issue 6

The `sinf()` and `sinl()` functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

System Interfaces

NAME
sinh, sinhf, sinhl — hyperbolic sine functions

SYNOPSIS
#include <math.h>
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);

DESCRIPTION
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.

These functions shall compute the hyperbolic sine of their argument \( x \).

An application wishing to check for error situations should set \( \text{errno} \) to zero and call \( \text{feclearexcept} \) (FE_ALL_EXCEPT) before calling these functions. On return, if \( \text{errno} \) is non-zero or \( \text{fetestexcept} \) (FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE
Upon successful completion, these functions shall return the hyperbolic sine of \( x \).

If the result would cause an overflow, a range error shall occur and \( \pm \text{HUGE}_\text{VAL}, \pm \text{HUGE}_\text{VALF}, \text{and} \pm \text{HUGE}_\text{VALL} \) (with the same sign as \( x \)) shall be returned as appropriate for the type of the function.

If \( x \) is NaN, a NaN shall be returned.

If \( x \) is \( \pm 0 \) or \( \pm \text{Inf} \), \( x \) shall be returned.

If \( x \) is subnormal, a range error may occur and \( x \) should be returned.

ERRORS
These functions shall fail if:

Range Error The result would cause an overflow.

If the integer expression (\( \text{math_errhandling} \& \text{MATH_ERRNO} \)) is non-zero, then \( \text{errno} \) shall be set to \( \text{[ERANGE]} \). If the integer expression (\( \text{math_errhandling} \& \text{MATH_ERREXCEPT} \)) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

Range Error The value \( x \) is subnormal.

If the integer expression (\( \text{math_errhandling} \& \text{MATH_ERRNO} \)) is non-zero, then \( \text{errno} \) shall be set to \( \text{[ERANGE]} \). If the integer expression (\( \text{math_errhandling} \& \text{MATH_ERREXCEPT} \)) is non-zero, then the underflow floating-point exception shall be raised.
EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
asinh(), cosh(), feclearexcept(), fetestexcept(), isnan(), tanh(), the Base Definitions volume of
IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions,
<math.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This
text was previously published in the APPLICATION USAGE section.

Issue 6
The sinh() and sinhl() functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
revised to align with the ISO/IEC 9899:1999 standard.
NAME
sinl — sine function

SYNOPSIS
#include <math.h>
long double sinl(long double x);

DESCRIPTION
Refer to sin().
NAME
sleep — suspend execution for an interval of time

SYNOPSIS
#include <unistd.h>

unsigned sleep(unsigned seconds);

DESCRIPTION
The sleep() function shall cause the calling thread to be suspended from execution until either
the number of realtime seconds specified by the argument seconds has elapsed or a signal is
delivered to the calling thread and its action is to invoke a signal-catching function or to
terminate the process. The suspension time may be longer than requested due to the scheduling
of other activity by the system.

If a SIGALRM signal is generated for the calling process during execution of sleep() and if the
SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether sleep() returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also
unspecified whether it remains pending after sleep() returns or it is discarded.

If a SIGALRM signal is generated for the calling process during execution of sleep(), except as a
result of a prior call to alarm(), and if the SIGALRM signal is not being ignored or blocked from
delivery, it is unspecified whether that signal has any effect other than causing sleep() to return.

If a signal-catching function interrupts sleep() and examines or changes either the time a
SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or
whether the SIGALRM signal is blocked from delivery, the results are unspecified.

If a signal-catching function interrupts sleep() and calls siglongjmp() or longjmp() to restore an
environment saved prior to the sleep() call, the action associated with the SIGALRM signal and
the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also
unspecified whether the SIGALRM signal is blocked, unless the process’ signal mask is restored
as part of the environment.

XSI Interactions between sleep() and any of setitimer(), ualarm(), or usleep() are unspecified.

RETURN VALUE
If sleep() returns because the requested time has elapsed, the value returned shall be 0. If sleep() returns due to delivery of a signal, the return value shall be the “unslept” amount (the requested
time minus the time actually slept) in seconds.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
There are two general approaches to the implementation of the sleep() function. One is to use the
alarm() function to schedule a SIGALRM signal and then suspend the process waiting for that
signal. The other is to implement an independent facility. This volume of IEEE Std 1003.1-2001
permits either approach.

In order to comply with the requirement that no primitive shall change a process attribute unless
explicitly described by this volume of IEEE Std 1003.1-2001, an implementation using SIGALRM
must carefully take into account any SIGALRM signal scheduled by previous alarm() calls, the
action previously established for SIGALRM, and whether SIGALRM was blocked. If a SIGALRM has been scheduled before the `sleep()` would ordinarily complete, the `sleep()` must be shortened to that time and a SIGALRM generated (possibly simulated by direct invocation of the signal-catching function) before `sleep()` returns. If a SIGALRM has been scheduled after the `sleep()` would ordinarily complete, it must be rescheduled for the same time before `sleep()` returns. The action and blocking for SIGALRM must be saved and restored.

Historical implementations often implement the SIGALRM-based version using `alarm()` and `pause()`. One such implementation is prone to infinite hangups, as described in `pause()`. Another such implementation uses the C-language `setjmp()` and `longjmp()` functions to avoid that window. That implementation introduces a different problem: when the SIGALRM signal interrupts a signal-catching function installed by the user to catch a different signal, the `longjmp()` aborts that signal-catching function. An implementation based on `sigprocmask()`, `alarm()`, and `sigsuspend()` can avoid these problems.

Despite all reasonable care, there are several very subtle, but detectable and unavoidable, differences between the two types of implementations. These are the cases mentioned in this volume of IEEE Std 1003.1-2001 where some other activity relating to SIGALRM takes place, and the results are stated to be unspecified. All of these cases are sufficiently unusual as not to be of concern to most applications.

See also the discussion of the term `realtime` in `alarm()`.

Since `sleep()` can be implemented using `alarm()`, the discussion about alarms occurring early under `alarm()` applies to `sleep()` as well.

Application writers should note that the type of the argument `seconds` and the return value of `sleep()` is `unsigned`. That means that a Strictly Conforming POSIX System Interfaces Application cannot pass a value greater than the minimum guaranteed value for `{UINT_MAX}`, which the ISO C standard sets as 65 535, and any application passing a larger value is restricting its portability. A different type was considered, but historical implementations, including those with a 16-bit `int` type, consistently use either `unsigned` or `int`.

Scheduling delays may cause the process to return from the `sleep()` function significantly after the requested time. In such cases, the return value should be set to zero, since the formula (requested time minus the time actually spent) yields a negative number and `sleep()` returns an `unsigned`.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`alarm()`, `getitimer()`, `nanosleep()`, `pause()`, `sigaction()`, `sigsetjmp()`, `ualarm()`, `usleep()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<unistd.h>`

**CHANGE HISTORY**

**Issue 5**

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.
NAME
snprintf — print formatted output

SYNOPSIS
#include <stdio.h>

int snprintf(char *restrict s, size_t n,
const char *restrict format, ...);

DESCRIPTION
Refer to fprintf().
NAME
sockatmark — determine whether a socket is at the out-of-band mark

SYNOPSIS
#include <sys/socket.h>
int sockatmark(int s);

DESCRIPTION
The sockatmark() function shall determine whether the socket specified by the descriptor s is at
the out-of-band data mark (see the System Interfaces volume of IEEE Std 1003.1-2001, Section
2.10.12, Socket Out-of-Band Data State). If the protocol for the socket supports out-of-band data
by marking the stream with an out-of-band data mark, the sockatmark() function shall return 1
when all data preceding the mark has been read and the out-of-band data mark is the first
element in the receive queue. The sockatmark() function shall not remove the mark from the
stream.

RETURN VALUE
Upon successful completion, the sockatmark() function shall return a value indicating whether
the socket is at an out-of-band data mark. If the protocol has marked the data stream and all data
preceding the mark has been read, the return value shall be 1; if there is no mark, or if data
precedes the mark in the receive queue, the sockatmark() function shall return 0. Otherwise, it
shall return a value of −1 and set errno to indicate the error.

ERRORS
The sockatmark() function shall fail if:
[EBADF] The s argument is not a valid file descriptor.
[ENOTTY] The s argument does not specify a descriptor for a socket.

EXAMPLES
None.

APPLICATION USAGE
The use of this function between receive operations allows an application to determine which
received data precedes the out-of-band data and which follows the out-of-band data.

There is an inherent race condition in the use of this function. On an empty receive queue, the
current read of the location might well be at the “mark”, but the system has no way of knowing
that the next data segment that will arrive from the network will carry the mark, and
sockatmark() will return false, and the next read operation will silently consume the mark.

Hence, this function can only be used reliably when the application already knows that the out-
of-band data has been seen by the system or that it is known that there is data waiting to be read
at the socket (via SIGURG or select()). See Section 2.10.11 (on page 61), Section 2.10.12 (on page
61), Section 2.10.14 (on page 62), and pselect() for details.

RATIONALE
The sockatmark() function replaces the historical SIOCATMARK command to ioctl() which
implemented the same functionality on many implementations. Using a wrapper function
follows the adopted conventions to avoid specifying commands to the ioctl() function, other
than those now included to support XSI STREAMS. The sockatmark() function could be
implemented as follows:
#include <sys/ioctl.h>
int sockatmark(int s)
{
```c
int val;
if (ioctl(s, SIOCATMARK, &val)==-1)
    return(-1);
return(val);

The use of [ENOTTY] to indicate an incorrect descriptor type matches the historical behavior of
SIOCATEMARK.

FUTURE DIRECTIONS
None.

SEE ALSO
pselect(), recv(), recvmsg(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

CHANGE HISTORY
```
NAME
socket — create an endpoint for communication

SYNOPSIS
#include <sys/socket.h>
int socket(int domain, int type, int protocol);

DESCRIPTION
The socket() function shall create an unbound socket in a communications domain, and return a
file descriptor that can be used in later function calls that operate on sockets.

The socket() function takes the following arguments:

- **domain**: Specifies the communications domain in which a socket is to be created.
- **type**: Specifies the type of socket to be created.
- **protocol**: Specifies a particular protocol to be used with the socket. Specifying a protocol
  of 0 causes socket() to use an unspecified default protocol appropriate for the
  requested socket type.

The domain argument specifies the address family used in the communications domain. The
address families supported by the system are implementation-defined.

Symbolic constants that can be used for the domain argument are defined in the <sys/socket.h>
header.

The type argument specifies the socket type, which determines the semantics of communication
over the socket. The following socket types are defined; implementations may specify additional
socket types:

- **SOCK_STREAM**: Provides sequenced, reliable, bidirectional, connection-mode byte
  streams, and may provide a transmission mechanism for out-of-band
  data.
- **SOCK_DGRAM**: Provides datagrams, which are connectionless-mode, unreliable messages
  of fixed maximum length.
- **SOCK_SEQPACKET**: Provides sequenced, reliable, bidirectional, connection-mode
  transmission paths for records. 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 part of more than one record. Record
  boundaries are visible to the receiver via the MSG_EOR flag.

If the protocol argument is non-zero, it shall specify a protocol that is supported by the address
family. If the protocol argument is zero, the default protocol for this address family and type shall
be used. The protocols supported by the system are implementation-defined.

The process may need to have appropriate privileges to use the socket() function or to create
some sockets.

RETURN VALUE
Upon successful completion, socket() shall return a non-negative integer, the socket file
descriptor. Otherwise, a value of −1 shall be returned and errno set to indicate the error.

ERRORS
The socket() function shall fail if:

- [EAFNOSUPPORT]
The implementation does not support the specified address family.
socket() No more file descriptors are available for this process.

[ENFILE] No more file descriptors are available for the system.

[EPROTONOSUPPORT] The protocol is not supported by the address family, or the protocol is not supported by the implementation.

[EPROTOTYPE] The socket type is not supported by the protocol.

The socket() function may fail if:

[EACCES] The process does not have appropriate privileges.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

EXAMPLES None.

APPLICATION USAGE 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.

The application can determine whether an address family is supported by trying to create a socket with domain set to the protocol in question.

RATIONALE None.

FUTURE DIRECTIONS None.

SEE ALSO accept(), bind(), connect(), getsockname(), getsockopt(), listen(), recv(), recvfrom(), recvmsg(), send(), sendmsg(), setsockopt(), shutdown(), socketpair(), the Base Definitions volume of IEEE Std 1003.1-2001, <netinet/in.h>, <sys/socket.h>

CHANGE HISTORY First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
socketpair — create a pair of connected sockets

SYNOPSIS
#include <sys/socket.h>
int socketpair(int domain, int type, int protocol,
               int socket_vector[2]);

DESCRIPTION
The socketpair() function shall create an unbound pair of connected sockets in a specified domain, of a specified type, under the protocol optionally specified by the protocol argument. The two sockets shall be identical. The file descriptors used in referencing the created sockets shall be returned in socket_vector[0] and socket_vector[1].

The socketpair() function takes the following arguments:

- **domain**: Specifies the communications domain in which the sockets are to be created.
- **type**: Specifies the type of sockets to be created.
- **protocol**: Specifies a particular protocol to be used with the sockets. Specifying a protocol of 0 causes socketpair() to use an unspecified default protocol appropriate for the requested socket type.
- **socket_vector**: Specifies a 2-integer array to hold the file descriptors of the created socket pair.

The type argument specifies the socket type, which determines the semantics of communications over the socket. The following socket types are defined; implementations may specify additional socket types:

- **SOCK_STREAM**: Provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.
- **SOCK_DGRAM**: Provides datagrams, which are connectionless-mode, unreliable messages of fixed maximum length.
- **SOCK_SEQPACKET**: Provides sequenced, reliable, bidirectional, connection-mode transmission paths for records. 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 part of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag.

If the protocol argument is non-zero, it shall specify a protocol that is supported by the address family. If the protocol argument is zero, the default protocol for this address family and type shall be used. The protocols supported by the system are implementation-defined.

The process may need to have appropriate privileges to use the socketpair() function or to create some sockets.

RETURN VALUE
Upon successful completion, this function shall return 0; otherwise, −1 shall be returned and errno set to indicate the error.

ERRORS
The socketpair() function shall fail if:

- [EAFNOSUPPORT]: The implementation does not support the specified address family.
socketpair()  

System Interfaces

43084 [EMFILE]  No more file descriptors are available for this process.
43085 [ENFILE]  No more file descriptors are available for the system.
43086 [EOPNOTSUPP]  The specified protocol does not permit creation of socket pairs.
43087 [EPROTONOSUPPORT]  The protocol is not supported by the address family, or the protocol is not supported by the implementation.
43088 [EPROTOTYPE]  The socket type is not supported by the protocol.
43091 The socketpair() function may fail if:
43092 [EACCES]  The process does not have appropriate privileges.
43093 [ENOBUFS]  Insufficient resources were available in the system to perform the operation.
43094 [ENOMEM]  Insufficient memory was available to fulfill the request.

EXAMPLES

None.

APPLICATION USAGE

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.

The socketpair() function is used primarily with UNIX domain sockets and need not be supported for other domains.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

socket(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/socket.h>

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.
NAME
sprintf — print formatted output

SYNOPSIS
#include <stdio.h>

int sprintf(char *restrict s, const char *restrict format, ...);

DESCRIPTION
Refer to fprintf().
NAME
sqrt, sqrtf, sqrtl — square root function

SYNOPSIS
#include <math.h>

double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);

DESCRIPTION
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.

These functions shall compute the square root of their argument \( x, \sqrt{x} \).

An application wishing to check for error situations should set \( \text{errno} \) to zero and call \( \text{feclearexcept}(\text{FE_ALL_EXCEPT}) \) before calling these functions. On return, if \( \text{errno} \) is non-zero or \( \text{fetestexcept}(\text{FE_INVALID} \mid \text{FE_DIVBYZERO} \mid \text{FE_OVERFLOW} \mid \text{FE_UNDERFLOW}) \) is non-zero, an error has occurred.

RETURN VALUE
Upon successful completion, these functions shall return the square root of \( x \).

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.

If \( x \) is NaN, a NaN shall be returned.

If \( x \) is ±0 or +Inf, \( x \) shall be returned.

If \( x \) is −Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS
These functions shall fail if:

Domain Error
The finite value of \( x \) is \(< -0 \), or \( x \) is \(-\infty \).

If the integer expression \( \text{math_errhandling} \& \text{MATH_ERRNO} \) is non-zero, then \( \text{errno} \) shall be set to \[ \text{EDOM} \]. If the integer expression \( \text{math_errhandling} \& \text{MATH_ERREXCEPT} \) is non-zero, then the invalid floating-point exception shall be raised.

EXAMPLES

Taking the Square Root of 9.0
#include <math.h>
...
double x = 9.0;
double result;
...
result = sqrt(x);
APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
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>, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This
text was previously published in the APPLICATION USAGE section.

Issue 6
The sqrtf() and sqrtl() functions are added for alignment with the ISO/IEC 9899:1999 standard.
The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are
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.
NAME
srand — pseudo-random number generator

SYNOPSIS
#include <stdlib.h>
void srand(unsigned seed);

DESCRIPTION
Refer to rand().
NAME
srand48 — seed the uniformly distributed double-precision pseudo-random number generator

SYNOPSIS
XSI
#include <stdlib.h>

void srand48(long seedval);

DESCRIPTION
Refer to drand48().
srandom()  

NAME
srandom — seed pseudo-random number generator

SYNOPSIS
XSI  

```c
#include <stdlib.h>

void srandom(unsigned seed);
```

DESCRIPTION
Refer to `initstate()`. 
NAME
sscanf — convert formatted input

SYNOPSIS
#include <stdio.h>
int sscanf(const char *restrict s, const char *restrict format, ...);

DESCRIPTION
Refer to fscanf().
NAME
stat — get file status

SYNOPSIS
#include <sys/stat.h>

int stat(const char *restrict path, struct stat *restrict buf);

DESCRIPTION
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.

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.

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.

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.

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.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to
indicate the error.

ERRORS
The stat() function shall fail if:

[EACCES] Search permission is denied for a component of the path prefix.

[EIO] An error occurred while reading from the file system.

[ELOOP] A loop exists in symbolic links encountered during resolution of the path
argument.

[ENAMETOOLONG]
The length of the path argument exceeds [PATH_MAX] or a pathname
component is longer than [NAME_MAX].

[ENOENT] A component of path does not name an existing file or path is an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

[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.

The stat() function may fail if:
Example 1: Obtaining File Status Information

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct stat buffer;
int status;

status = stat("/home/cnd/mod1", &buffer);
```

Example 2: Getting Directory Information

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <locale.h>
#include <langinfo.h>
#include <stdio.h>
#include <stdint.h>

struct dirent *dp;
struct stat statbuf;
struct passwd *pwd;
struct group *grp;
struct tm *tm;
char datestring[256];

/* Loop through directory entries. */
while ((dp = readdir(dir)) != NULL) {
    /* Get entry’s information. */
    if (stat(dp->d_name, &statbuf) == -1) {
        continue;
    }
    // Further processing of entry information...
}
```
/* Print out type, permissions, and number of links. */
printf("%10.10s", sperm (statbuf.st_mode));
printf("%4d", statbuf.st_nlink);
/* Print out owner’s name if it is found using getpwuid(). */
if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
  printf(" %-8.8s", pwd->pw_name);
else
  printf(" %-8d", statbuf.st_uid);
/* Print out group name if it is found using getgrgid(). */
if ((grp = getgrgid(statbuf.st_gid)) != NULL)
  printf(" %-8.8s", grp->gr_name);
else
  printf(" %-8d", statbuf.st_gid);
/* Print size of file. */
printf(" %9jd", (intmax_t)statbuf.st_size);
tm = localtime(&statbuf.st_mtime);
/* Get localized date string. */
strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);
printf(" %s %s
", datestring, dp->d_name);

APPLICATION USAGE
None.

RATIONALE
The intent of the paragraph describing “additional or alternate file access control mechanisms”
is to allow a secure implementation where a process with a label that does not dominate the
file’s label cannot perform a stat() function. This is not related to read permission; a process with
a label that dominates the file’s label does not need read permission. An implementation that
supports write-up operations could fail fstat() function calls even though it has a valid file
descriptor open for writing.

FUTURE DIRECTIONS
None.

SEE ALSO
fstat(), lstat(), readlink(), symlink(), the Base Definitions volume of IEEE Std 1003.1-2001,
<sys/stat.h>, <sys/types.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
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.
The [EIO] mandatory error condition is added.

The [ELOOP] mandatory error condition is added.

The [EOVERFLOW] mandatory error condition is added. This change is to support large files.

The [ENAMETOOLONG] and the second [EOVERFLOW] optional error conditions are added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Details are added regarding the treatment of symbolic links.
- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

The restrict keyword is added to the stat() prototype for alignment with the ISO/IEC 9899:1999 standard.
statvfs()  

NAME  
statvfs — get file system information  

SYNOPSIS  

XSI  
```
#include <sys/statvfs.h>

int statvfs(const char *restrict path, struct statvfs *restrict buf);
```  

DESCRIPTION  
Refer to fstatvfs().
NAME
stderr, stdin, stdout — standard I/O streams

SYNOPSIS
#include <stdio.h>
extern FILE *stderr, *stdin, *stdout;

DESCRIPTION
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

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.

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.

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:

STDIN_FILENO Standard input value, stdin. Its value is 0.
STDOUT_FILENO Standard output value, stdout. Its value is 1.
STDERR_FILENO Standard error value, stderr. Its value is 2.

The stderr stream is expected to be open for reading and writing.

RETURN VALUE
None.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fclose(), feof(), ferror(), fileno(), fopen(), fread(), fseek(), gets(), popen(), printf(), putc(),
puts(), read(), scanf(), setbuf(), setvbuf(), tmpfile(), ungetc(), vprintf(), the Base Definitions
volume of IEEE Std 1003.1-2001, <stdio.h>, <unistd.h>
stdin

43399 CHANGE HISTORY
43400 First released in Issue 1.
43401 Issue 6
43402 Extensions beyond the ISO C standard are marked.
43403 A note that stderr is expected to be open for reading and writing is added to the DESCRIPTION.
NAME
strncasecmp, strcasecmp — case-insensitive string comparisons

SYNOPSIS
#include <strings.h>

int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);

DESCRIPTION
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.

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.

RETURN VALUE
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.

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.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <strings.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
NAME
strcat — concatenate two strings

SYNOPSIS
#include <string.h>

char *strcat(char *restrict s1, const char *restrict s2);

DESCRIPTION
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 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.

RETURN VALUE
The strcat() function shall return s1; no return value is reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strncat(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The strcat() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
strchr — string scanning operation

SYNOPSIS
#include <string.h>
char *strchr(const char *s, int c);

DESCRIPTION
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 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.

RETURN VALUE
Upon completion, strchr() shall return a pointer to the byte, or a null pointer if the byte was not found.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strchr(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.
NAME

strcmp — compare two strings

SYNOPSIS

```c
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

DESCRIPTION

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 `strcmp()` function shall compare the string pointed to by `s1` to the string pointed to by `s2`.

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 bytes (both interpreted as type `unsigned char`) that differ in the strings being compared.

RETURN VALUE

Upon completion, `strcmp()` shall return an integer greater than, equal to, or less than 0, if the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2`, respectively.

ERRORS

No errors are defined.

EXAMPLES

Checking a Password Entry

The following example compares the information read from standard input to the value of the name of the user entry. If the `strcmp()` function returns 0 (indicating a match), a further check will be made to see if the user entered the proper old password. The `crypt()` function shall encrypt the old password entered by the user, using the value of the encrypted password in the `passwd` structure as the salt. If this value matches the value of the encrypted `passwd` in the structure, the entered password `oldpasswd` is the correct user's password. Finally, the program encrypts the new password so that it can store the information in the `passwd` structure.

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
...
int valid_change;
struct passwd *p;
char user[100];
char oldpasswd[100];
char newpasswd[100];
char savepasswd[100];
...
if (strcmp(p->pw_name, user) == 0) {
    if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
        strcpy(savepasswd, crypt(newpasswd, user));
        p->pw_passwd = savepasswd;
        valid_change = 1;
    }
} else {
```
    fprintf(stderr, "Old password is not valid\n");
    
...
NAME
  strcoll — string comparison using collating information

SYNOPSIS
  #include <string.h>
  int strcoll(const char *s1, const char *s2);

DESCRIPTION
  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

  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.

  The strcoll() function shall not change the setting of errno if successful.

  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.

RETURN VALUE
  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.

ERRORS
  The strcoll() function may fail if:

  [EINVAL] The s1 or s2 arguments contain characters outside the domain of the collating
  sequence.

EXAMPLES
  Comparing Nodes
  The following example uses an application-defined function, node_compare(), to compare two
  nodes based on an alphabetical ordering of the string field.

  #include <string.h>
  ...
  struct node { /* These are stored in the table. */
    char *string;
    int length;
  };
  ...
  int node_compare(const void *node1, const void *node2)
  {
    return strcoll(((const struct node *)node1)->string,
                   ((const struct node *)node2)->string);
  }
  ...

APPLICATION USAGE
  The strxfrm() and strcmp() functions should be used for sorting large lists.
RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
\texttt{strcmp()}, \texttt{strxfrm()}, the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<string.h>}

CHANGE HISTORY
First released in Issue 3.

Issue 5
The DESCRIPTION is updated to indicate that \texttt{errno} does not change if the function is successful.

Issue 6
Extensions beyond the ISO C standard are marked.
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:
\begin{itemize}
  \item The [EINVAL] optional error condition is added.
\end{itemize}
An example is added.
**NAME**
strcpy — copy a string

**SYNOPSIS**
```
#include <string.h>
char *strcpy(char *restrict s1, const char *restrict s2);
```

**DESCRIPTION**
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 `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.

**RETURN VALUE**
The `strcpy()` function shall return `s1`; no return value is reserved to indicate an error.

**ERRORS**
No errors are defined.

**EXAMPLES**

### Initializing a String
The following example copies the string "----------" into the `permstring` variable.
```
#include <string.h>
...
static char permstring[11];
...
strcpy(permstring, "----------");
```

### Storing a Key and Data
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` *)
```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
...
/* Structure used to read data and store it. */
struct element {
    char *key;
    char *data;
};
struct element *tbl, *curtbl;
char *key, *data;
int count;
...
void dbfree(struct element *, int);
```
... if ((curtbl->key = malloc(strlen(key) + 1)) == NULL) {
    perror("malloc"); dbfree(tbl, count); return NULL;
}
strcpy(curtbl->key, key);

if ((curtbl->data = malloc(strlen(data) + 1)) == NULL) {
    perror("malloc"); free(curtbl->key); dbfree(tbl, count); return NULL;
}
strcpy(curtbl->data, data);
...

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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strncpy(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The strcpy() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
strcspn — get the length of a complementary substring

SYNOPSIS
#include <string.h>
size_t strcspn(const char *s1, const char *s2);

DESCRIPTION
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

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.

RETURN VALUE
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.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strspn(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The RETURN VALUE section is updated to indicate that strcspn() returns the length of s1, and
not s1 itself as was previously stated.

Issue 6
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.
NAME
strdup — duplicate a string

SYNOPSIS
XSI
#include <string.h>
char *strdup(const char *s1);

DESCRIPTION
The strdup() function shall return a pointer to a new string, which is a duplicate of the string
pointed to by s1. The returned pointer can be passed to free(). A null pointer is returned if the
new string cannot be created.

RETURN VALUE
The strdup() function shall return a pointer to a new string on success. Otherwise, it shall return
a null pointer and set errno to indicate the error.

ERRORS
The strdup() function may fail if:
[ENOMEM] Storage space available is insufficient.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
free(), malloc(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
**NAME**
strerror, strerror_r — get error message string

**SYNOPSIS**
```
#include <string.h>

char *strerror(int errnum);
```
```
int strerror_r(int errnum, char *strerrbuf, size_t buflen);
```

**DESCRIPTION**
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.

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.

The string pointed to shall not be modified by the application, but may be overwritten by a subsequent call to `strerror()` or `perror()`.

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.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-2001 calls `strerror()`.

The `strerror()` function shall not change the setting of `errno` if successful.

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`.

The `strerror()` function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

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`.

**RETURN VALUE**
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.

Upon successful completion, `strerror_r()` shall return 0. Otherwise, an error number shall be returned to indicate the error.

**ERRORS**
These functions may fail if:
```
[EINVAL] The value of `errnum` is not a valid error number.
```

The `strerror_r()` function may fail if:
```
[ERANGE] Insufficient storage was supplied via `strerrbuf` and `buflen` to contain the generated message string.
```
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
perror(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 3.

Issue 5
The DESCRIPTION is updated to indicate that errno is not changed if the function is successful.
A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6
Extensions beyond the ISO C standard are marked.
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the fact that errno may be set is added.
- The [EINVAL] optional error condition is added.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The strerror_r() function is added in response to IEEE PASC Interpretation 1003.1c #39.
The strerror_r() function is marked as part of the Thread-Safe Functions option.
strfmon()

NAME

strfmon — convert monetary value to a string

SYNOPSIS

XSI

```
#include <monetary.h>

ssize_t strfmon(char *restrict s, size_t maxsize,
    const char *restrict format, ...);
```

DESCRIPTION

The strfmon() function shall place characters into the array pointed to by s as controlled by the
string pointed to by format. No more than maxsize bytes are placed into the array.

The format is a character string, beginning and ending in its initial state, if any, that contains two
types of objects: plain characters, which are simply copied to the output stream, and conversion
specifications, each of which shall result in the fetching of zero or more arguments which are
converted and formatted. The results are undefined if there are insufficient arguments for the
format. If the format is exhausted while arguments remain, the excess arguments are simply
ignored.

The application shall ensure that a conversion specification consists of the following sequence:

- A ‘%’ character
- Optional flags
- Optional field width
- Optional left precision
- Optional right precision
- A required conversion specifier character that determines the conversion to be performed

Flags

One or more of the following optional flags can be specified to control the conversion:

- An ‘=’ followed by a single character f which is used as the numeric fill character. In
  order to work with precision or width counts, the fill character shall be a single byte
  character; if not, the behavior is undefined. The default numeric fill character is the
  <space>. This flag does not affect field width filling which always uses the <space>.
  This flag is ignored unless a left precision (see below) is specified.
- Do not format the currency amount with grouping characters. The default is to insert
  the grouping characters if defined for the current locale.
- Specify the style of representing positive and negative currency amounts. Only one of
  ‘+’ or ‘ ’ may be specified. If ‘+’ is specified, the locale’s equivalent of ‘+’ and ‘−’
  are used (for example, in the U.S., the empty string if positive and ‘−’ if negative). If
  ‘ ’ is specified, negative amounts are enclosed within parentheses. If neither flag is
  specified, the ‘+’ style is used.
- Suppress the currency symbol from the output conversion.
- Specify the alignment. If this flag is present the result of the conversion is left-justified
  (padded to the right) rather than right-justified. This flag shall be ignored unless a field
  width (see below) is specified.
Field Width

$w$  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.

Left Precision

$\#n$  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 ‘=’ flag above).

If grouping has not been suppressed with the ‘ˆ’ 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.

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.

Right Precision

$.p$  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.

Conversion Specifier Characters

The conversion specifier characters and their meanings are:

$i$  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.

$n$  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.

%  Convert to a ‘%’; no argument is converted. The entire conversion specification shall be %.

Locale Information

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: 1995 standard.

If the value of maxsize is greater than {SSIZE_MAX}, the result is implementation-defined.
RETURN VALUE

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
including the terminating null byte. Otherwise, −1 shall be returned, the contents of the array are
unspecified, and errno shall be set to indicate the error.

ERRORS

The strfmon() function shall fail if:

[E2BIG] Conversion stopped due to lack of space in the buffer.

EXAMPLES

Given a locale for the U.S. and the values 123.45, −123.45, and 3456.781, the following output
might be produced. Square brackets ("[ ]") are used in this example to delimit the output.

%n                [$123.45]    Default formatting
[−$123.45]
[$3,456.78]

%11n              [    $123.45]  Right align within an 11-character field
[    −$123.45]
[    $3,456.78]

%#5n               [ $    123.45]  Aligned columns for values up to 99999
[−$    123.45]
[    $3,456.78]

%=*#5n             [ ***$123.45]  Specify a fill character
[−***$123.45]
[    $*3,456.78]

%0#5n              [ $000123.45]  Fill characters do not use grouping
[−$000123.45]
[    $03,456.78]

%5n                [ $    123.45]  Disable the grouping separator
[−$    123.45]
[    $3456.78]

%5.0n               [ $    123]  Round off to whole units
[−$    123]
[    $3457]

%#5.4n              [ $    123.4500]  Increase the precision
[−$    123.4500]
[    $3456.7810]

%(#5n               [ $(    123.45]  Use an alternative pos/neg style
[($    123.45)]
[$(    3,456.78]

%! (#5n             [ 123.45]  Disable the currency symbol
[ 123.45]
[ 3,456.78]

%−14#5.4n           [ $    123.4500 ]  Left-justify the output
[−$    123.4500 ]
[    $3,456.7810 ]
%14#5.4n  [ $ 123.4500]  Corresponding right-justified output
[ -$ 123.4500]
[ $ 3,456.7810]

See also the EXAMPLES section in *fprintf*().

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
Lowercase conversion characters are reserved for future standards use and uppercase for implementation-defined use.

**SEE ALSO**
*fprintf*(), *localeconv*(), the Base Definitions volume of IEEE Std 1003.1-2001, `<monetary.h>`

**CHANGE HISTORY**
First released in Issue 4.

**Issue 5**
Moved from ENHANCED I18N to BASE.
The [ENOSYS] error is removed.
A sentence is added to the DESCRIPTION warning about values of `maxsize` that are greater than [SSIZE_MAX].

**Issue 6**
The DESCRIPTION is updated to avoid use of the term "must" for application requirements.
The `restrict` keyword is added to the `strfmon()` prototype for alignment with the ISO/IEC 9899:1999 standard.
The EXAMPLES section is reworked, clarifying the output format.
NAME
strftimetime — convert date and time to a string

SYNOPSIS
#include <time.h>

size_t strftime(char *restrict s, size_t maxsize, const char *restrict format, const struct tm *restrict timeptr);

DESCRIPTION
The functionality described on this reference page is aligned with the ISO C standard. Any
cflict between the requirements described here and the ISO C standard is unintentional. This
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.

Local timezone information is used as though strftime() called tzset().
The following conversion specifications are supported:
%a Replaced by the locale’s abbreviated weekday name. [tm_wday]
%A Replaced by the locale’s full weekday name. [tm_wday]
%b Replaced by the locale’s abbreviated month name. [tm_mon]
%B Replaced by the locale’s full month name. [tm_mon]
%c Replaced by the locale’s appropriate date and time representation. (See the Base
Definitions volume of IEEE Std 1003.1-2001, <time.h>.)
%c Replaced by the year divided by 100 and truncated to an integer, as a decimal number
[00,99]. [tm_year]
%d Replaced by the day of the month as a decimal number [01,31]. [tm_mday]
%D Equivalent to %m/%d/%y. [tm_mon, tm_mday, tm_year]
%e Replaced by the day of the month as a decimal number [1,31]; a single digit is preceded
by a space. [tm_mday]
%f Equivalent to %Y-%m-%d (the ISO 8601:2000 standard date format). [tm_year, tm_mon,
tm_mday]
%g Replaced by the last 2 digits of the week-based year (see below) as a decimal number
[00,99]. [tm_year, tm_wday, tm_yday]
%G Replaced by the week-based year (see below) as a decimal number (for example, 1977).
[tm_year, tm_wday, tm_yday]
System Interfaces

strftime()

Equivalent to \%b. [tm_mon]

\%H
Replaced by the hour (24-hour clock) as a decimal number [00,23]. [tm_hour]

\%I
Replaced by the hour (12-hour clock) as a decimal number [01,12]. [tm_hour]

\%j
Replaced by the day of the year as a decimal number [001,366]. [tm_yday]

\%m
Replaced by the month as a decimal number [01,12]. [tm_mon]

\%M
Replaced by the minute as a decimal number [00,59]. [tm_min]

\%n
Replaced by a <newline>.

\%p
Replaced by the locale's equivalent of either a.m. or p.m. [tm_hour]

\%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]

\%R
Replaced by the time in 24-hour notation (\%H:\%M). [tm_hour, tm_min]

\%S
Replaced by the second as a decimal number [00,60]. [tm_sec]

\%t
Replaced by a <tab>.

\%T
Replaced by the time (\%H:\%M:\%S). [tm_hour, tm_min, tm_sec]

\%u
Replaced by the weekday as a decimal number [1,7], with 1 representing Monday. [tm_wday]

\%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]

\%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]

\%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]

\%w
Replaced by the weekday as a decimal number [0,6], with 0 representing Sunday. [tm_wday]

\%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]

\%x
Replaced by the locale's appropriate date representation. (See the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>.)

\%X
Replaced by the locale's appropriate time representation. (See the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>.)

\%y
Replaced by the last two digits of the year as a decimal number [00,99]. [tm_year]

\%Y
Replaced by the year as a decimal number (for example, 1997). [tm_year]

\%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]
strftime( )  

System Interfaces

44056   %Z    Replaced by the timezone name or abbreviation, or by no bytes if no timezone information exists. [%m_isdst]
44057
44058   %%    Replaced by %.
44059
44060 If a conversion specification does not correspond to any of the above, the behavior is undefined.

Modified Conversion Specifiers

44061 Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist for the current locale (see ERA in the Base Definitions volume of IEEE Std 1003.1-2001, Section 7.3.5, LC_TIME), the behavior shall be as if the unmodified conversion specification were used.

44066    %Ec    Replaced by the locale’s alternative appropriate date and time representation.
44067    %EC    Replaced by the name of the base year (period) in the locale’s alternative representation.
44068    %Ex    Replaced by the locale’s alternative date representation.
44069    %EX    Replaced by the locale’s alternative time representation.
44070    %Ey    Replaced by the offset from %EC (year only) in the locale’s alternative representation.
44071    %EY    Replaced by the full alternative year representation.
44072    %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.
44073    %Od    Replaced by the day of the month, using the locale’s alternative numeric symbols, filled as needed with leading spaces.
44077    %OE    Replaced by the day of the month, using the locale’s alternative numeric symbols, filled as needed with leading spaces.
44078    %OH    Replaced by the hour (24-hour clock) using the locale’s alternative numeric symbols.
44079    %OI    Replaced by the hour (12-hour clock) using the locale’s alternative numeric symbols.
44080    %OM    Replaced by the month using the locale’s alternative numeric symbols.
44081    %OS    Replaced by the minutes using the locale’s alternative numeric symbols.
44082    %OS    Replaced by the seconds using the locale’s alternative numeric symbols.
44083    %OU    Replaced by the weekday as a number in the locale’s alternative representation (Monday=1).
44084    %OU    Replaced by the week number of the year (Monday as the first day of the week) using the locale’s alternative numeric symbols.
44085    %OV    Replaced by the week number of the year (Monday as the first day of the week) using the locale’s alternative numeric symbols.
44086    %OV    Replaced by the week number of the year (Monday as the first day of the week) using the locale’s alternative numeric symbols.
44087    %OW    Replaced by the number of the weekday (Sunday=0) using the locale’s alternative numeric symbols.
44088    %OW    Replaced by the week number of the year (Monday as the first day of the week) using the locale’s alternative numeric symbols.
44089    %OW    Replaced by the week number of the year (Monday as the first day of the week) using the locale’s alternative numeric symbols.
44090    %OW    Replaced by the year (offset from %C) using the locale’s alternative numeric symbols.
44091    %g, %G, and %V give values according to the ISO 8601:2000 standard week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th,
which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, %G is replaced by 1998 and %V is replaced by 53. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, %G is replaced by 1998 and %V is replaced by 01.

If a conversion specifier is not one of the above, the behavior is undefined.

**RETURN VALUE**

If the total number of resulting bytes including the terminating null byte is not more than maxsize, strftime() shall return the number of bytes placed into the array pointed to by s, not including the terminating null byte. Otherwise, 0 shall be returned and the contents of the array are unspecified.

**ERRORS**

No errors are defined.

**EXAMPLES**

### Getting a Localized Date String

The following example first sets the locale to the user’s default. The locale information will be used in the nl_langinfo() and strftime() functions. The nl_langinfo() function returns the localized date string which specifies how the date is laid out. The strftime() function takes this information and, using the tm structure for values, places the date and time information into datestring.

```c
#include <time.h>
#include <locale.h>
#include <langinfo.h>
...
struct tm *tm;
char datestring[256];
...
setlocale (LC_ALL, "");
...
strftime (datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);
...
```

### APPLICATION USAGE

The range of values for %S is [00,60] rather than [00,59] to allow for the occasional leap second.

Some of the conversion specifications are duplicates of others. They are included for compatibility with nlctime() and nl_ascxtime(), which were published in Issue 2.

Applications should use %Y (4-digit years) in preference to %y (2-digit years).

In the C locale, the E and O modifiers are ignored and the replacement strings for the following specifiers are:

- %a The first three characters of %A.
- %A One of Sunday, Monday, ..., Saturday.
- %b The first three characters of %B.
- %B One of January, February, ..., December.
- %c Equivalent to %a %b %e %T %Y.
strftime()

%p One of AM or PM.
%r Equivalent to %I:%M:%S %p.
%x Equivalent to %m/%d/%y.
%x Equivalent to %T.
%Z Implementation-defined.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
asctime(), clock(), ctime(), difftime(), getdate(), gmtime(), localtime(), mktime(), strftime(), time(),
tzset(), utime(), Base Definitions volume of IEEE Std 1003.1-2001, Section 7.3.5, LC_TIME,
<time.h>

CHANGE HISTORY
First released in Issue 3.

Issue 5
The description of %OV is changed to be consistent with %V and defines Monday as the first day
of the week.
The description of %Oy is clarified.

Issue 6
Extensions beyond the ISO C standard are marked.
The Open Group Corrigendum U033/8 is applied. The %V conversion specifier is changed from
"Otherwise, it is week 53 of the previous year, and the next week is week 1" to "Otherwise, it is
the last week of the previous year, and the next week is week 1".
The following new requirements on POSIX implementations derive from alignment with the
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
date utility.
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.
A new example is added.
NAME
strlen — get string length

SYNOPSIS
#include <string.h>
size_t strlen(const char *s);

DESCRIPTION
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 `strlen()` function shall compute the number of bytes in the string to which `s` points, not including the terminating null byte.

RETURN VALUE
The `strlen()` function shall return the length of `s`; no return value shall be reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES

Getting String Lengths
The following example sets the maximum length of `key` and `data` by using `strlen()` to get the lengths of those strings.

```
#include <string.h>
...
struct element {
    char *key;
    char *data;
};
...
char *key, *data;
int len;
*keylength = *datalength = 0;
...
if ((len = strlen(key)) > *keylength)
    *keylength = len;
if ((len = strlen(data)) > *datalength)
    *datalength = len;
```

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.
strlen()

**SEE ALSO**
The Base Definitions volume of IEEE Std 1003.1-2001, `<string.h>`

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**
The RETURN VALUE section is updated to indicate that `strlen()` returns the length of `s`, and not `s` itself as was previously stated.
strncasecmp() — case-insensitive string comparison

SYNOPSIS

```
#include <strings.h>

int strncasecmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

Refer to `strcasecmp()`.
strncat()

NAME
strncat — concatenate a string with part of another

SYNOPSIS
#include <string.h>
char *strncat(char *restrict s1, const char *restrict s2, size_t n);

DESCRIPTION
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 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.

RETURN VALUE
The strncat() function shall return s1; no return value shall be reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strcat(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The strncat() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
strncmp — compare part of two strings

SYNOPSIS
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n);

DESCRIPTION
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 strncmp() 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.

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.

RETURN VALUE
Upon successful completion, strncmp() 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.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strcmp(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.
strncpy() — copy part of a string

#include <string.h>

char *strncpy(char *restrict s1, const char *restrict s2, size_t n);

The strcpy() 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.

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.

The strcpy() function shall return \( s1 \); no return value is reserved to indicate an error.

No errors are defined.

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

If there is no null byte in the first \( n \) bytes of the array pointed to by \( s2 \), the result is not null-terminated.

None.

First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
strpbrk — scan a string for a byte

SYNOPSIS
#include <string.h>
char *strpbrk(const char *s1, const char *s2);

DESCRIPTION
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

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.

RETURN VALUE
Upon successful completion, strpbrk() shall return a pointer to the byte or a null pointer if no
byte from s2 occurs in s1.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strchr(), strrchr(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
strptime — date and time conversion

SYNOPSIS
XSI
#include <time.h>

char *strptime(const char *restrict buf, const char *restrict format,
                struct tm *restrict tm);

DESCRIPTION
The strftime() function shall convert the character string pointed to by buf
to values which are stored in the tm structure pointed to by tm, using the
format specified by format.

The format is composed of zero or more directives. Each directive is composed
of one of the following: one or more white-space characters (as specified by
isspace()); an ordinary character (neither ‘%’ nor a white-space character);
or a conversion specification. Each conversion specification is composed of a
‘%’ character followed by a conversion character which specifies the
replacement required. The application shall ensure that there is white-space or
other non-alphanumeric characters between any two conversion specifications. The following conversion
specifications are supported:

%a  The day of the week, using the locale's weekday names; either the abbreviated or full
    name may be specified.
%A  Equivalent to %a.
%b  The month, using the locale's month names; either the abbreviated or full name may be
    specified.
%B  Equivalent to %b.
%c  Replaced by the locale's appropriate date and time representation.
%C  The century number [00,99]; leading zeros are permitted but not required.
%d  The day of the month [01,31]; leading zeros are permitted but not required.
%D  The date as %m/%d/%y.
%e  Equivalent to %d.
%e  Equivalent to %b.
%H  The hour (24-hour clock) [00,23]; leading zeros are permitted but not required.
%I  The hour (12-hour clock) [01,12]; leading zeros are permitted but not required.
%j  The day number of the year [001,366]; leading zeros are permitted but not required.
%m  The month number [01,12]; leading zeros are permitted but not required.
%M  The minute [00,59]; leading zeros are permitted but not required.
%n  Any white space.
%p  The locale's equivalent of a.m or p.m.
%r  12-hour clock time using the AM/PM notation if _fmt_ampm_ is not an empty string in
    the LC_TIME portion of the current locale; in the POSIX locale, this shall be equivalent
to %I:%M:%S %p.
%R  The time as %H:%M.
%S  The seconds [00,60]; leading zeros are permitted but not required.
%t  Any white space.
%T  The time as %H:%M:%S.
%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.
%w  The weekday as a decimal number [0,6], with 0 representing Sunday; leading zeros are permitted but not required.
%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.
%x  The date, using the locale’s date format.
%x  The time, using the locale’s time format.
%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.

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.)

%Y  The year, including the century (for example, 1988).
%%  Replaced by %.

Modified Conversion Specifiers

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.

%Ec  The locale’s alternative appropriate date and time representation.
%EC  The name of the base year (period) in the locale’s alternative representation.
%Ex  The locale’s alternative date representation.
%EX  The locale’s alternative time representation.
%Ey  The offset from %EC (year only) in the locale’s alternative representation.
%EY  The full alternative year representation.
%Od  The day of the month using the locale’s alternative numeric symbols; leading zeros are permitted but not required.
%Oe  Equivalent to %Od.
%OH  The hour (24-hour clock) using the locale’s alternative numeric symbols.
%OI  The hour (12-hour clock) using the locale’s alternative numeric symbols.
%Om  The month using the locale’s alternative numeric symbols.
%OM  The minutes using the locale’s alternative numeric symbols.
strptime()  

The seconds using the locale's alternative numeric symbols.

The week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.

The number of the weekday (Sunday=0) using the locale's alternative numeric symbols.

The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.

The year (offset from %C) using the locale's alternative numeric symbols.

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.

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.

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.

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, strptime() fails and no more characters are scanned.

RETURN VALUE

Upon successful completion, strptime() shall return a pointer to the character following the last character parsed. Otherwise, a null pointer shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Several “equivalent to” formats and the special processing of white-space characters are provided in order to ease the use of identical format strings for strftime() and strptime().

Applications should use %Y (4-digit years) in preference to %y (2-digit years).

It is unspecified whether multiple calls to strptime() using the same tm structure will update the current contents of the structure or overwrite all contents of the structure. Conforming applications should make a single call to strptime() with a format and all data needed to completely specify the date and time being converted.

RATIONALE

None.
FUTURE DIRECTIONS
The `strptime()` function is expected to be mandatory in the next version of this volume of IEEE Std 1003.1-2001.

SEE ALSO
`scanf()`, `strftime()`, `time()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<time.h>`

CHANGE HISTORY
First released in Issue 4.

Issue 5
Moved from ENHANCED I18N to BASE.
The [ENOSYS] error is removed.
The exact meaning of the `%y` and `%%oy` specifiers is clarified in the DESCRIPTION.

Issue 6
The Open Group Corrigendum U033/5 is applied. The `%r` specifier description is reworded.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The `restrict` keyword is added to the `strptime()` prototype for alignment with the ISO/IEC 9899:1999 standard.
The Open Group Corrigendum U047/2 is applied.
The DESCRIPTION is updated to use the terms “conversion specifier” and “conversion specification” for consistency with `strftime()`.
strrchr() — string scanning operation

#include <string.h>

char *strrchr(const char *s, int c);

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.

Upon successful completion, strrchr() shall return a pointer to the byte or a null pointer if c does not occur in the string.

No errors are defined.

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.

```
#include <string.h>
...
const char *name;
char *basename;
...
basename = strrchr(name, '/') + 1;
...
```

Finding the Base Name of a File

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strchr(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
strspn — get length of a substring

SYNOPSIS
#include <string.h>

size_t strspn(const char *s1, const char *s2);

DESCRIPTION
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 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.

RETURN VALUE
The strspn() function shall return the length of s1; no return value is reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strcspn(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The RETURN VALUE section is updated to indicate that strspn() returns the length of s, and not s itself as was previously stated.
NAME
strstr — find a substring

SYNOPSIS
#include <string.h>
char *strstr(const char *s1, const char *s2);

DESCRIPTION
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
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.

RETURN VALUE
Upon successful completion, strstr() shall return a pointer to the located string or a null pointer
if the string is not found.
If s2 points to a string with zero length, the function shall return s1.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strchr(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>

CHANGE HISTORY
First released in Issue 3. Included for alignment with the ANSI C standard.
NAME
strtod, strtof, strtold — convert a string to a double-precision number

SYNOPSIS
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr);
float strtof(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);

DESCRIPTION
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.

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:

1. An initial, possibly empty, sequence of white-space characters (as specified by isspace())
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string

Then they shall attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part
- One of INF or INFINITY, ignoring case
- One of NAN or NAN(n-char-sequence_opt), ignoring case in the NAN part, where:

  n-char-sequence:
  digit
  nondigit
  n-char-sequence digit
  n-char-sequence nondigit

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.

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) 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.

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 [EINV].

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].

These functions shall fail if:

[ERANGE] The value to be returned would cause overflow or underflow.

These functions may fail if:

[EINV] No conversion could be performed.

None.

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:

```c
int what_kind_of_number (char *s)
{
    char *endp;
    double d;
    long l;
    d = strtod(s, &endp);
    if (s != endp && *endp == '\0')
        printf("It's a float with value %g\n", d);
    else
    {
        l = strtol(s, &endp, 0);
        if (s != endp && *endp == '\0')
            printf("It's an integer with value %ld\n", l);
        else
            return 1;
    }
    return 0;
}
```

If the function is called with:

```c
what_kind_of_number ("0x10")
```

an ISO/IEC 9899:1990 standard-compliant library will result in the function printing:

```plaintext
It's an integer with value 16
```

With the ISO/IEC 9899:1999 standard, the result is:

```plaintext
It's a float with value 16
```

The change in behavior is due to the inclusion of floating-point numbers in hexadecimal notation without requiring that either a decimal point or the binary exponent be present.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`isspace()`, `localeconv()`, `scanf()`, `setlocale()`, `strtol()`, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, `<float.h>`, `<stdlib.h>`

**CHANGE HISTORY**

First released in Issue 1. Derived from Issue 1 of the SVID.
The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the `[EINVAL]` optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The `strtod()` function is updated.
- The `strtof()` and `strtold()` functions are added.
- The DESCRIPTION is extensively revised.

NAME
strtoimax, strtoumax — convert string to integer type

SYNOPSIS
#include <inttypes.h>

intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr,
int base);

uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr,
int base);

DESCRIPTION
The functionality described on this reference page is aligned with the ISO C standard. Any
collision between the requirements described here and the ISO C standard is unintentional. This

These functions shall be equivalent to the `strtol()`, `strtoll()`, `strtoul()`, and `strtoull()` functions,
extcept that the initial portion of the string shall be converted to `intmax_t` and `uintmax_t`
representation, respectively.

RETURN VALUE
These functions shall return the converted value, if any.

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]`.

ERRORS
These functions shall fail if:

[ERANGE] The value to be returned is not representable.

These functions may fail if:

[EINVAL] The value of `base` is not supported.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
`strtol()`, `strtoul()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<inttypes.h>`

CHANGE HISTORY
NAME

strtok, strtok_r — split string into tokens

SYNOPSIS

```c
#include <string.h>

char *strtok(char *restrict s1, const char *restrict s2);

char *strtok_r(char *restrict s, const char *restrict sep, char **restrict lasts);
```

DESCRIPTION

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.

A sequence of calls to `strtok()` breaks the string pointed to by `s1` into a sequence of tokens, each of which is delimited by a byte from the string pointed to by `s2`. The first call in the sequence has `s1` as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by `s2` may be different from call to call.

The first call in the sequence searches the string pointed to by `s1` for the first byte that is not contained in the current separator string pointed to by `s2`. If no such byte is found, then there are no tokens in the string pointed to by `s1` and `strtok()` shall return a null pointer. If such a byte is found, it is the start of the first token.

The `strtok()` function then searches from there for a byte that is contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by `s1`, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a null byte, which terminates the current token. The `strtok()` function saves a pointer to the following byte, from which the next search for a token shall start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-2001 calls `strtok()`.

For `strtok_r()`:

The `strtok_r()` function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The `strtok_r()` function considers the null-terminated string `s` as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string `sep`. The argument `lasts` points to a user-provided pointer which points to stored information necessary for `strtok_r()` to continue scanning the same string.

In the first call to `strtok_r()`, `s` points to a null-terminated string, `sep` to a null-terminated string of separator characters, and the value pointed to by `lasts` is ignored. The `strtok_r()` function shall return a pointer to the first character of the first token, write a null character into `s` immediately following the returned token, and update the pointer to which `lasts` points.

In subsequent calls, `s` is a NULL pointer and `lasts` shall be unchanged from the previous call so that subsequent calls shall move through the string `s`, returning successive tokens until no tokens remain. The separator string `sep` may be different from call to call. When no token remains in `s`, a NULL pointer shall be returned.
RETURN VALUE
Upon successful completion, `strtok()` shall return a pointer to the first byte of a token. Otherwise, if there is no token, `strtok()` shall return a null pointer.

TSF The `strtok_r()` function shall return a pointer to the token found, or a NULL pointer when no token is found.

ERRORS
No errors are defined.

EXAMPLES

Searching for Word Separators
The following example searches for tokens separated by <space>s.

```
#include <string.h>
...
char *token;
char *line = "LINE TO BE SEPARATED";
char *search = " ";
/* Token will point to "LINE". */
token = strtok(line, search);
/* Token will point to "TO". */
token = strtok(NULL, search);
```

Breaking a Line
The following example uses `strtok()` to break a line into two character strings separated by any combination of <space>s, <tab>s, or <newline>s.

```
#include <string.h>
...
struct element {
    char *key;
    char *data;
};
...
char line[LINE_MAX];
char *key, *data;
... key = strtok(line, " \n");
data = strtok(NULL, " \n");
```

APPLICATION USAGE
The `strtok_r()` function is thread-safe and stores its state in a user-supplied buffer instead of possibly using a static data area that may be overwritten by an unrelated call from another thread.

RATIONALE
The `strtok()` function searches for a separator string within a larger string. It returns a pointer to the last substring between separator strings. This function uses static storage to keep track of the current string position between calls. The new function, `strtok_r()`, takes an additional argument, `lasts`, to keep track of the current position in the string.
**FUTURE DIRECTIONS**
None.

**SEE ALSO**
The Base Definitions volume of IEEE Std 1003.1-2001, `<string.h>`

**CHANGE HISTORY**
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**
The `strtok_r()` function is included for alignment with the POSIX Threads Extension.
A note indicating that the `strtok()` function need not be reentrant is added to the DESCRIPTION.

**Issue 6**
Extensions beyond the ISO C standard are marked.
The `strtok_r()` function is marked as part of the Thread-Safe Functions option.
In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.
The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.
The `restrict` keyword is added to the `strtok()` and `strtok_r()` prototypes for alignment with the ISO/IEC 9899:1999 standard.
NAME
strtol, strtoll — convert a string to a long integer

SYNOPSIS
#include <stdlib.h>

long strtol(const char *restrict str, char **restrict endptr, int base);
long long strtoll(const char *restrict str, char **restrict endptr,
      int base)

DESCRIPTION
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

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:

1. An initial, possibly empty, sequence of white-space characters (as specified by isspace( ))
2. A subject sequence interpreted as an integer represented in some radix determined by the
   value of base
3. A final string of one or more unrecognized characters, including the terminating null byte
   of the input string.

Then they shall attempt to convert the subject sequence to an integer, and return the result.

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.

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.

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.

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.

In other than the C or POSIX locales, other implementation-defined subject sequences may be
accepted.
The `strtol()` function shall not change the setting of `errno` if successful.

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 should set `errno` to 0, then call `strtol()` or `strtoll()`, then check `errno`.

### RETURN VALUE
Upon successful completion, these functions shall return the converted value, if any. 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, `[LONG_MIN]`, `[LONG_MAX]`, `[LLONG_MIN]`, or `[LLONG_MAX]` shall be returned (according to the sign of the value), and `errno` set to `[ERANGE]`.

### ERRORS
These functions shall fail if:

- `[ERANGE]` The value to be returned is not representable.

These functions may fail if:

- `[EINVAL]` The value of `base` is not supported.

### EXAMPLES
None.

### APPLICATION USAGE
None.

### RATIONALE
None.

### FUTURE DIRECTIONS
None.

### SEE ALSO
`isalpha()`, `scanf()`, `strtof()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`

### CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**
The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

**Issue 6**
Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the `[EINVAL]` optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The `strtol()` prototype is updated.
- The `strtoll()` function is added.
NAME
strtol — convert a string to a double-precision number

SYNOPSIS
#include <stdlib.h>

long double strtol(const char *restrict nptr, char **restrict endptr);

DESCRIPTION
Refer to _strtod_.

NAME
strtol — convert a string to a long integer

SYNOPSIS
#include <stdlib.h>

long long strtol(const char *restrict str, char **restrict endptr,
                int base);

DESCRIPTION
Refer to strtol().
NAME
strtoul, strtoull — convert a string to an unsigned long

SYNOPSIS
#include <stdlib.h>

unsigned long strtoul(const char *restrict str,
        char **restrict endptr, int base);
unsigned long long strtoull(const char *restrict str,
        char **restrict endptr, int base);

DESCRIPTION
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

These functions shall convert the initial portion of the string pointed to by str to a type unsigned
long and unsigned long long representation, respectively. First, they decompose the input
string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by isspace( ))
2. A subject sequence interpreted as an integer represented in some radix determined by the
   value of base
3. A final string of one or more unrecognized characters, including the terminating null byte
   of the input string

Then they shall attempt to convert the subject sequence to an unsigned integer, and return the
result.

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.

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.

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.

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.
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 shall be stored in the object pointed to by endptr, provided that endptr is not a null pointer.

The `strtoul()` function shall not change the setting of `errno` if successful.

Since 0, [ULONG_MAX], and [ULLONG_MAX] are returned on error and are also valid returns on success, an application wishing to check for error situations should set `errno` to 0, then call `strtoul()` or `strtoull()`, then check `errno`.

RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. 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, [ULONG_MAX] or [ULLONG_MAX] shall be returned and `errno` set to [ERANGE].

ERRORS

These functions shall fail if:

- [EINVAL] The value of base is not supported.
- [ERANGE] The value to be returned is not representable.

These functions may fail if:

- [EINVAL] No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

`isalpha()`, `scanf()`, `strtod()`, `strtol()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Issue 5

The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] error condition is added for when the value of base is not supported.

In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.
The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The `strtoul()` prototype is updated.
- The `strtoull()` function is added.
NAME
stroumax — convert a string to an integer type

SYNOPSIS
#include <inttypes.h>

uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr,
        int base);

DESCRIPTION
Refer to strtoimax().
NAME
strxfrm — string transformation

SYNOPSIS
#include <string.h>

size_t strxfrm(char *restrict s1, const char *restrict s2, size_t n);

DESCRIPTION
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

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.

The strxfrm() function shall not change the setting of errno if successful.

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.

RETURN VALUE
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.

On error, strxfrm() may set errno but no return value is reserved to indicate an error.

ERRORS
The strxfrm() function may fail if:

[EINVAL] The string pointed to by the s2 argument contains characters outside the
domain of the collating sequence.

EXAMPLES
None.

APPLICATION USAGE
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).

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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strcmp(), strcoll(), the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>
**CHANGE HISTORY**

First released in Issue 3. Included for alignment with the ISO C standard.

**Issue 5**

The DESCRIPTION is updated to indicate that `errno` does not change if the function is successful.

**Issue 6**

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The `strxfrm()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
swab — swap bytes

SYNOPSIS
#include <unistd.h>

void swab(const void *restrict src, void *restrict dest, ssize_t nbytes);

DESCRIPTION
The swab() function shall copy nbytes bytes, which are pointed to by src, to the object pointed to by dest, exchanging adjacent bytes. The nbytes argument should be even. If nbytes is odd, swab() copies and exchanges nbytes–1 bytes and the disposition of the last byte is unspecified. If copying takes place between objects that overlap, the behavior is undefined. If nbytes is negative, swab() does nothing.

RETURN VALUE
None.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The restrict keyword is added to the swab() prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
swapcontext — swap user context

SYNOPSIS
#define <ucontext.h>

int swapcontext(ucontext_t *restrict oucp,
const ucontext_t *restrict ucp);

DESCRIPTION
Refer to makecontext().
NAME
swprintf — print formatted wide-character output

SYNOPSIS
#include <stdio.h>
#include <wchar.h>

int swprintf(wchar_t *restrict ws, size_t n,
             const wchar_t *restrict format,…);

DESCRIPTION
Refer to fprintf().
NAME
swscanf — convert formatted wide-character input

SYNOPSIS
#include <stdio.h>
#include <wchar.h>

int swscanf(const wchar_t *restrict ws, const wchar_t *restrict format, ...);

DESCRIPTION
Refer to fscanf().
NAME
symlink — make a symbolic link to a file

SYNOPSIS
#include <unistd.h>

int symlink(const char *path1, const char *path2);

DESCRIPTION
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).

The string pointed to by path1 shall be treated only as a character string and shall not be
validated as a pathname.

If the symlink() function fails for any reason other than [EIO], any file named by path2 shall be
unaffected.

RETURN VALUE
Upon successful completion, symlink() shall return 0; otherwise, it shall return −1 and set errno to
indicate the error.

ERRORS
The symlink() function shall fail if:

[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.

[EEXIST] The path2 argument names an existing file or symbolic link.

[EIO] An I/O error occurs while reading from or writing to the file system.

[ELOOP] A loop exists in symbolic links encountered during resolution of the path2
argument.

[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}.

[ENOENT] A component of path2 does not name an existing file or path2 is an empty
string.

[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.

[ENOTDIR] A component of the path prefix of path2 is not a directory.

[EROFS] The new symbolic link would reside on a read-only file system.

The symlink() function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
resolution of the path2 argument.

[ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the path2...
argument, the length of the substituted pathname string exceeded
(PATH_MAX) bytes (including the terminating null byte), or the length of the
string pointed to by path1 exceeded [SYMLINK_MAX].

EXAMPLES
None.

APPLICATION USAGE
Like a hard link, a symbolic link allows a file to have multiple logical names. The presence of a
hard link guarantees the existence of a file, even after the original name has been removed. A
symbolic link provides no such assurance; in fact, the file named by the path1 argument need not
exist when the link is created. A symbolic link can cross file system boundaries.

Normal permission checks are made on each component of the symbolic link pathname during
its resolution.

RATIONALE
Since IEEE Std 1003.1-2001 does not require any association of file times with symbolic links,
there is no requirement that file times be updated by symlink().

FUTURE DIRECTIONS
None.

SEE ALSO
lchown(), link(), lstat(), open(), readlink(), unlink(), the Base Definitions volume of
IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 4, Version 2.
Moved from X/OPEN UNIX extension to BASE.

The following changes were made to align with the IEEE P1003.1a draft standard:
• The DESCRIPTION text is updated.
• The [ELOOP] optional error condition is added.
NAME
sync — schedule file system updates

SYNOPSIS
#include <unistd.h>
void sync(void);

DESCRIPTION
The sync() function shall cause all information in memory that updates file systems to be
scheduled for writing out to all file systems.
The writing, although scheduled, is not necessarily complete upon return from sync().

RETURN VALUE
The sync() function shall not return a value.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fsync(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.
NAME
sysconf — get configurable system variables

SYNOPSIS
#include <unistd.h>
long sysconf(int name);

DESCRIPTION
The sysconf() function provides a method for the application to determine the current value of a configurable system limit or option (variable). Support for some system variables is dependent on implementation options (as indicated by the margin codes in the following table). Where an implementation option is not supported, the variable need not be supported.

The name argument represents the system variable to be queried. The following table lists the minimal set of system variables from <limits.h> or <unistd.h> that can be returned by sysconf(), and the symbolic constants defined in <unistd.h> that are the corresponding values used for name. Support for some configuration variables is dependent on implementation options (see shading and margin codes in the table below). Where an implementation option is not supported, the variable need not be supported.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value of Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIO_LISTIO_MAX</td>
<td>_SC_AIO_LISTIO_MAX</td>
</tr>
<tr>
<td>AIO_MAX</td>
<td>_SC_AIO_MAX</td>
</tr>
<tr>
<td>AIO_PRIO DELTA_MAX</td>
<td>_SC_AIO_PRIO DELTA_MAX</td>
</tr>
<tr>
<td>ARG_MAX</td>
<td>_SC_ARG_MAX</td>
</tr>
<tr>
<td>ATEXIT_MAX</td>
<td>_SC_ATEXIT_MAX</td>
</tr>
<tr>
<td>BC_BASE_MAX</td>
<td>_SC_BC_BASE_MAX</td>
</tr>
<tr>
<td>BC_DIM_MAX</td>
<td>_SC_BC_DIM_MAX</td>
</tr>
<tr>
<td>BC_SCALE_MAX</td>
<td>_SC_BC_SCALE_MAX</td>
</tr>
<tr>
<td>BC_STRING_MAX</td>
<td>_SC_BC_STRING_MAX</td>
</tr>
<tr>
<td>CHILD_MAX</td>
<td>_SC_CHILD_MAX</td>
</tr>
<tr>
<td>COLL_WEIGHTS_MAX</td>
<td>_SC_COLL_WEIGHTS_MAX</td>
</tr>
<tr>
<td>CLOCK_TICKS/second</td>
<td>_SC_CLK_TCK</td>
</tr>
<tr>
<td>DELAYTIMER_MAX</td>
<td>_SC_DELAYTIMER_MAX</td>
</tr>
<tr>
<td>EXPR_NEST_MAX</td>
<td>_SC_EXPR_NEST_MAX</td>
</tr>
<tr>
<td>HOST_NAME_MAX</td>
<td>_SC_HOST_NAME_MAX</td>
</tr>
<tr>
<td>IOV_MAX</td>
<td>_SC_IOV_MAX</td>
</tr>
<tr>
<td>LINE_MAX</td>
<td>_SC_LINE_MAX</td>
</tr>
<tr>
<td>LOGIN_NAME_MAX</td>
<td>_SC_LOGIN_NAME_MAX</td>
</tr>
<tr>
<td>GROUPS_MAX</td>
<td>_SC_NGROUPS_MAX</td>
</tr>
<tr>
<td>MQ_OPEN_MAX</td>
<td>_SC_MQ_OPEN_MAX</td>
</tr>
<tr>
<td>MQ_PRIO_MAX</td>
<td>_SC_MQ_PRIO_MAX</td>
</tr>
</tbody>
</table>

Maximum size of getgrgid_r() and getgrnam_r() data buffers
Maximum size of getpwnam_r() and getpwnam_r() data buffers
<table>
<thead>
<tr>
<th>Variable</th>
<th>Value of Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[OPEN_MAX]</td>
<td>_SC_OPEN_MAX</td>
</tr>
<tr>
<td>_POSIX_ADVISORY_INFO</td>
<td>_SC_ADVISORY_INFO</td>
</tr>
<tr>
<td>_POSIX_BARRIERS</td>
<td>_SC_BARRIERS</td>
</tr>
<tr>
<td>_POSIXASYNCHRONOUS_IO</td>
<td>_SCASYNCHRONOUS_IO</td>
</tr>
<tr>
<td>_POSIX_CLOCK_SELECTION</td>
<td>_SC_CLOCK_SELECTION</td>
</tr>
<tr>
<td>_POSIX_CPU_TIME</td>
<td>_SC_CPU_TIME</td>
</tr>
<tr>
<td>_POSIX_FILE_LOCKING</td>
<td>_SC_FILE_LOCKING</td>
</tr>
<tr>
<td>_POSIX_FSYNC</td>
<td>_SC_FSYNC</td>
</tr>
<tr>
<td>_POSIX_JOB_CONTROL</td>
<td>_SC_JOB_CONTROL</td>
</tr>
<tr>
<td>_POSIX_MAPPED_FILES</td>
<td>_SC_MAPPED_FILES</td>
</tr>
<tr>
<td>_POSIX_MEMLOCK</td>
<td>_SC_MEMLOCK</td>
</tr>
<tr>
<td>_POSIX_MEMLOCK_RANGE</td>
<td>_SC_MEMLOCK_RANGE</td>
</tr>
<tr>
<td>_POSIX_MEMORY_PROTECTION</td>
<td>_SC_MEMORY_PROTECTION</td>
</tr>
<tr>
<td>_POSIX_MESSAGE_PASSING</td>
<td>_SC_MESSAGE_PASSING</td>
</tr>
<tr>
<td>_POSIX_MONOTONIC_CLOCK</td>
<td>_SC_MONOTONIC_CLOCK</td>
</tr>
<tr>
<td>_POSIX_MULTI_PROCESS</td>
<td>_SC_MULTI_PROCESS</td>
</tr>
<tr>
<td>_POSIX_PRIORITIZED_IO</td>
<td>_SC_PRIORITIZED_IO</td>
</tr>
<tr>
<td>_POSIX_PRIORITY_SCHEDULING</td>
<td>_SC_PRIORITY_SCHEDULING</td>
</tr>
<tr>
<td>_POSIX_READER_WRITER_LOCKS</td>
<td>_SC_READER_WRITER_LOCKS</td>
</tr>
<tr>
<td>_POSIX_REALTIME_SIGNALS</td>
<td>_SC_REALTIME_SIGNALS</td>
</tr>
<tr>
<td>_POSIX_REGEXP</td>
<td>_SC_REGEXP</td>
</tr>
<tr>
<td>_POSIX_SAVED_IDS</td>
<td>_SC_SAVED_IDS</td>
</tr>
<tr>
<td>_POSIX_SEMAPHORES</td>
<td>_SC_SEMAPHORES</td>
</tr>
<tr>
<td>_POSIX_SHARED_MEMORY_OBJECTS</td>
<td>_SC_SHARED_MEMORY_OBJECTS</td>
</tr>
<tr>
<td>_POSIX_SHELL</td>
<td>_SC_SHELL</td>
</tr>
<tr>
<td>_POSIX_SPAWN</td>
<td>_SC_SPAWN</td>
</tr>
<tr>
<td>_POSIX_SPIN_LOCKS</td>
<td>_SC_SPIN_LOCKS</td>
</tr>
<tr>
<td>_POSIX_SPORADIC_SERVER</td>
<td>_SC_SPORADIC_SERVER</td>
</tr>
<tr>
<td>_POSIX_SYNCHRONIZED_IO</td>
<td>_SC_SYNCHRONIZED_IO</td>
</tr>
<tr>
<td>_POSIX_THREAD_ATTR_STACKADDR</td>
<td>_SC_THREAD_ATTR_STACKADD</td>
</tr>
<tr>
<td>_POSIX_THREAD_ATTR_STACKSIZE</td>
<td>_SC_THREAD_ATTR_STACKSIZE</td>
</tr>
<tr>
<td>_POSIX_THREAD_CPUTIME</td>
<td>_SC_THREAD_CPUTIME</td>
</tr>
<tr>
<td>_POSIX_THREAD_PRIO_INHERIT</td>
<td>_SC_THREAD_PRIO_INHERIT</td>
</tr>
<tr>
<td>_POSIX_THREAD_PRIO_PROTECT</td>
<td>_SC_THREAD_PRIO_PROTECT</td>
</tr>
<tr>
<td>_POSIX_THREAD_PRIORITY_SCHEDULING</td>
<td>_SC_THREAD_PRIORITY_SCHEDULING</td>
</tr>
<tr>
<td>_POSIX_THREAD_PROCESS_SHARED</td>
<td>_SC_THREAD_PROCESS_SHARED</td>
</tr>
<tr>
<td>_POSIX_THREAD_SAFE_FUNCTIONS</td>
<td>_SC_THREAD_SAFE_FUNCTIONS</td>
</tr>
<tr>
<td>_POSIX_THREAD_SPORADIC_SERVER</td>
<td>_SC_THREAD_SPORADIC_SERVER</td>
</tr>
<tr>
<td>_POSIX_THREADS</td>
<td>_SC_THREADS</td>
</tr>
<tr>
<td>_POSIX_TIMEOUTS</td>
<td>_SC_TIMEOUTS</td>
</tr>
<tr>
<td>_POSIX_TIMERS</td>
<td>_SC_TIMERS</td>
</tr>
<tr>
<td>_POSIX_TRACE</td>
<td>_SC_TRACE</td>
</tr>
<tr>
<td>_POSIX_TRACE_EVENT_FILTER</td>
<td>_SC_TRACE_EVENT_FILTER</td>
</tr>
<tr>
<td>_POSIX_TRACE_INHERIT</td>
<td>_SC_TRACE_INHERIT</td>
</tr>
<tr>
<td>_POSIX_TRACE_LOG</td>
<td>_SC_TRACE_LOG</td>
</tr>
<tr>
<td>_POSIX_TYPED_MEMORY_OBJECTS</td>
<td>_SC_TYPED_MEMORY_OBJECTS</td>
</tr>
<tr>
<td>Variable</td>
<td>Value of Name</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td><code>_POSIX_VERSION</code></td>
<td><code>SC_VERSION</code></td>
</tr>
<tr>
<td><code>_POSIX_V6_ILP32_OFF32</code></td>
<td><code>SC_V6_ILP32_OFF32</code></td>
</tr>
<tr>
<td><code>_POSIX_V6_ILP32_OFFBIG</code></td>
<td><code>SC_V6_ILP32_OFFBIG</code></td>
</tr>
<tr>
<td><code>_POSIX_V6_LP64_OFF64</code></td>
<td><code>SC_V6_LP64_OFF64</code></td>
</tr>
<tr>
<td><code>_POSIX_V6_LPBIG_OFFBIG</code></td>
<td><code>SC_V6_LPBIG_OFFBIG</code></td>
</tr>
<tr>
<td><code>_POSIX2_C_BIND</code></td>
<td><code>SC_2_C_BIND</code></td>
</tr>
<tr>
<td><code>_POSIX2_C_DEV</code></td>
<td><code>SC_2_C_DEV</code></td>
</tr>
<tr>
<td><code>_POSIX2_C_VERSION</code></td>
<td><code>SC_2_C_VERSION</code></td>
</tr>
<tr>
<td><code>_POSIX2_CHAR_TERM</code></td>
<td><code>SC_2_CHAR_TERM</code></td>
</tr>
<tr>
<td><code>_POSIX2_FORT_DEV</code></td>
<td><code>SC_2_FORT_DEV</code></td>
</tr>
<tr>
<td><code>_POSIX2_FORT_RUN</code></td>
<td><code>SC_2_FORT_RUN</code></td>
</tr>
<tr>
<td><code>_POSIX2_LOCALEDEF</code></td>
<td><code>SC_2_LOCALEDEF</code></td>
</tr>
<tr>
<td><code>_POSIX2_PBS</code></td>
<td><code>SC_2_PBS</code></td>
</tr>
<tr>
<td><code>_POSIX2_PBS_ACCOUNTING</code></td>
<td><code>SC_2_PBS_ACCOUNTING</code></td>
</tr>
<tr>
<td><code>_POSIX2_PBS_LOCATE</code></td>
<td><code>SC_2_PBS_LOCATE</code></td>
</tr>
<tr>
<td><code>_POSIX2_PBS_MESSAGE</code></td>
<td><code>SC_2_PBS_MESSAGE</code></td>
</tr>
<tr>
<td><code>_POSIX2_PBS_TRACK</code></td>
<td><code>SC_2_PBS_TRACK</code></td>
</tr>
<tr>
<td><code>_POSIX2_SW_DEV</code></td>
<td><code>SC_2_SW_DEV</code></td>
</tr>
<tr>
<td><code>_POSIX2_UPE</code></td>
<td><code>SC_2_UPE</code></td>
</tr>
<tr>
<td><code>_POSIX2_VERSION</code></td>
<td><code>SC_2_VERSION</code></td>
</tr>
<tr>
<td><code>_REGEX_VERSION</code></td>
<td><code>SC_REGEX_VERSION</code></td>
</tr>
<tr>
<td><code>XSI_PAGE_SIZE</code></td>
<td><code>SC_PAGE_SIZE</code></td>
</tr>
<tr>
<td><code>XSI_PAGESIZE</code></td>
<td><code>SC_PAGESIZE</code></td>
</tr>
<tr>
<td><code>XSI_PTHREAD_DESTRUCTOR_ITERATIONS</code></td>
<td><code>SC_PTHREAD_DESTRUCTOR_ITERATIONS</code></td>
</tr>
<tr>
<td><code>XSI_PTHREAD_KEYS_MAX</code></td>
<td><code>SC_PTHREAD_KEYS_MAX</code></td>
</tr>
<tr>
<td><code>XSI_PTHREAD_STACK_MIN</code></td>
<td><code>SC_PTHREAD_STACK_MIN</code></td>
</tr>
<tr>
<td><code>XSI_PTHREAD_THREADS_MAX</code></td>
<td><code>SC_PTHREAD_THREADS_MAX</code></td>
</tr>
<tr>
<td><code>XSI_RE_DUP_MAX</code></td>
<td><code>SC_RE_DUP_MAX</code></td>
</tr>
<tr>
<td><code>XSI_RTSIG_MAX</code></td>
<td><code>SC_RTSIG_MAX</code></td>
</tr>
<tr>
<td><code>XSI_SEM_NSEMS_MAX</code></td>
<td><code>SC_SEM_NSEMS_MAX</code></td>
</tr>
<tr>
<td><code>XSI_SEM_VALUE_MAX</code></td>
<td><code>SC_SEM_VALUE_MAX</code></td>
</tr>
<tr>
<td><code>XSI_SIGQUEUE_MAX</code></td>
<td><code>SC_SIGQUEUE_MAX</code></td>
</tr>
<tr>
<td><code>XSI_STREAM_MAX</code></td>
<td><code>SC_STREAM_MAX</code></td>
</tr>
<tr>
<td><code>XSI_SYMLOOP_MAX</code></td>
<td><code>SC_SYMLOOP_MAX</code></td>
</tr>
<tr>
<td><code>XSI_TIMER_MAX</code></td>
<td><code>SC_TIMER_MAX</code></td>
</tr>
<tr>
<td><code>XSI_TTY_NAME_MAX</code></td>
<td><code>SC_TTY_NAME_MAX</code></td>
</tr>
<tr>
<td><code>XSI_TZNAME_MAX</code></td>
<td><code>SC_TZNAME_MAX</code></td>
</tr>
<tr>
<td><code>XSI_XBS5_ILP32_OFF32</code></td>
<td><code>SC_XBS5_ILP32_OFF32</code></td>
</tr>
<tr>
<td><code>XSI_XBS5_ILP32_OFFBIG</code></td>
<td><code>SC_XBS5_ILP32_OFFBIG</code></td>
</tr>
<tr>
<td><code>XSI_XBS5_LP64_OFF64</code></td>
<td><code>SC_XBS5_LP64_OFF64</code></td>
</tr>
<tr>
<td><code>XSI_XBS5_LPBIG_OFFBIG</code></td>
<td><code>SC_XBS5_LPBIG_OFFBIG</code></td>
</tr>
<tr>
<td><code>XSI_XOPEN_CRYPT</code></td>
<td><code>SC_XOPEN_CRYPT</code></td>
</tr>
<tr>
<td><code>XSI_XOPEN_ENH_I18N</code></td>
<td><code>SC_XOPEN_ENH_I18N</code></td>
</tr>
</tbody>
</table>
System Interfaces

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value of Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSI _XOPEN_LEGACY</td>
<td>_SC_XOPEN_LEGACY</td>
</tr>
<tr>
<td>_XOPEN_REALTIME</td>
<td>_SC_XOPEN_REALTIME</td>
</tr>
<tr>
<td>_XOPEN_REALTIME_THREADS</td>
<td>_SC_XOPEN_REALTIME_THREADS</td>
</tr>
<tr>
<td>_XOPEN_SHM</td>
<td>_SC_XOPEN_SHM</td>
</tr>
<tr>
<td>_XOPEN_UNIX</td>
<td>_SC_XOPEN_UNIX</td>
</tr>
<tr>
<td>_XOPEN_VERSION</td>
<td>_SC_XOPEN_VERSION</td>
</tr>
<tr>
<td>_XOPEN_XCU_VERSION</td>
<td>_SC_XOPEN_XCU_VERSION</td>
</tr>
</tbody>
</table>

**RETURN VALUE**

If `name` is an invalid value, `sysconf()` shall return −1 and set `errno` to indicate the error. If the variable corresponding to `name` has no limit, `sysconf()` shall return −1 without changing the value of `errno`. Note that indefinite limits do not imply infinite limits; see `<limits.h>`.

Otherwise, `sysconf()` shall return the current variable value on the system. The value returned shall not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's `<limits.h>` or `<unistd.h>`. The value shall not change during the lifetime of the calling process.

**ERRORS**

The `sysconf()` function shall fail if:

- [EINVAL] The value of the `name` argument is invalid.

**EXAMPLES**

None.

**APPLICATION USAGE**

As −1 is a permissible return value in a successful situation, an application wishing to check for error situations should set `errno` to 0, then call `sysconf()`, and, if it returns −1, check to see if `errno` is non-zero.

If the value of `sysconf(_SC_2_VERSION)` is not equal to the value of the `_POSIX2_VERSION` symbolic constant, the utilities available via `system()` or `popen()` might not behave as described in the Shell and Utilities volume of IEEE Std 1003.1-2001. This would mean that the application is not running in an environment that conforms to the Shell and Utilities volume of IEEE Std 1003.1-2001. Some applications might be able to deal with this, others might not.

However, the functions defined in this volume of IEEE Std 1003.1-2001 continue to operate as specified, even if `sysconf(_SC_2_VERSION)` reports that the utilities no longer perform as specified.

**RATIONALE**

This functionality was added in response to requirements of application developers and of system vendors who deal with many international system configurations. It is closely related to `pathconf()` and `fpathconf()`.

Although a conforming application can run on all systems by never demanding more resources than the minimum values published in this volume of IEEE Std 1003.1-2001, it is useful for that application to be able to use the actual value for the quantity of a resource available on any given system. To do this, the application makes use of the value of a symbolic constant in `<limits.h>` or `<unistd.h>`.

However, once compiled, the application must still be able to cope if the amount of resource available is increased. To that end, an application may need a means of determining the quantity of a resource, or the presence of an option, at execution time.
Two examples are offered:

1. Applications may wish to act differently on systems with or without job control. Applications vendors who wish to distribute only a single binary package to all instances of a computer architecture would be forced to assume job control is never available if it were to rely solely on the `<unistd.h>` value published in this volume of IEEE Std 1003.1-2001.

2. International applications vendors occasionally require knowledge of the number of clock ticks per second. Without these facilities, they would be required to either distribute their applications partially in source form or to have 50 Hz and 60 Hz versions for the various countries in which they operate.

It is the knowledge that many applications are actually distributed widely in executable form that leads to this facility. If limited to the most restrictive values in the headers, such applications would have to be prepared to accept the most limited environments offered by the smallest microcomputers. Although this is entirely portable, there was a consensus that they should be able to take advantage of the facilities offered by large systems, without the restrictions associated with source and object distributions.

During the discussions of this feature, it was pointed out that it is almost always possible for an application to discern what a value might be at runtime by suitably testing the various functions themselves. And, in any event, it could always be written to adequately deal with error returns from the various functions. In the end, it was felt that this imposed an unreasonable level of complication and sophistication on the application writer.

This runtime facility is not meant to provide ever-changing values that applications have to check multiple times. The values are seen as changing no more frequently than once per system initialization, such as by a system administrator or operator with an automatic configuration program. This volume of IEEE Std 1003.1-2001 specifies that they shall not change within the lifetime of the process.

Some values apply to the system overall and others vary at the file system or directory level. The latter are described in `pathconf()`.

Note that all values returned must be expressible as integers. String values were considered, but the additional flexibility of this approach was rejected due to its added complexity of implementation and use.

Some values, such as `{PATH_MAX}`, are sometimes so large that they must not be used to, say, allocate arrays. The `sysconf()` function returns a negative value to show that this symbolic constant is not even defined in this case.

Similar to `pathconf()`, this permits the implementation not to have a limit. When one resource is infinite, returning an error indicating that some other resource limit has been reached is conforming behavior.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`confstr()`, `pathconf()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<limits.h>`, `<unistd.h>`, the Shell and Utilities volume of IEEE Std 1003.1-2001, `getconf`

**CHANGE HISTORY**

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.
The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

The _XBS_ variables and name values are added to the table of system variables in the DESCRIPTION. These are all marked EX.

The symbol CLK_TCK is obsolescent and removed. It is replaced with the phrase “clock ticks per second”.

The symbol [PASS_MAX] is removed.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Table entries are added for the following variables: _SC_REGEX, _SC_SHELL, _SC_REGEX_VERSION, _SC_SYMLOOP_MAX.

The following sysconf() variables and their associated names are added for alignment with IEEE Std 1003.1d-1999:

- _POSIX_ADVISORY_INFO
- _POSIX_CPUTIME
- _POSIX_SPAWN
- _POSIX_SPORADIC_SERVER
- _POSIX_THREAD_CPUTIME
- _POSIX_THREAD_SPORADIC_SERVER
- _POSIX_TIMEOUTS

The following changes are made to the DESCRIPTION for alignment with IEEE Std 1003.1j-2000:

- A statement expressing the dependency of support for some system variables on implementation options is added.

- 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

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

The following sysconf() variables and their associated names are added for alignment with IEEE Std 1003.1q-2000:

- _POSIX_TRACE
- _POSIX_TRACE_EVENT_FILTER
- _POSIX_TRACE_INHERIT
- _POSIX_TRACE_LOG
The macros associated with the c89 programming models are marked LEGACY, and new equivalent macros associated with c99 are introduced.
NAME
syslog — log a message

SYNOPSIS
XSI
#include <syslog.h>

void syslog(int priority, const char *message, ... /* argument */);

DESCRIPTION
Refer to closelog().
NAME
system — issue a command

SYNOPSIS
#include <stdlib.h>
int system(const char *command);

DESCRIPTION
If command is a null pointer, the system() function shall determine whether the host environment
has a command processor. If command is not a null pointer, the system() function shall pass the
string pointed to by command to that command processor to be executed in an implementation-
defined manner; this might then cause the program calling system() to behave in a non-
conforming manner or to terminate.

The environment of the executed command shall be as if a child process were created using
fork(), and the child process invoked the sh utility using execl() as follows:

    execl(<shell path>, "sh", ":c", command, (char *)0);

where <shell path> is an unspecified pathname for the sh utility.

The system() function shall ignore the SIGINT and SIGQUIT signals, and shall block the
SIGCHLD signal, while waiting for the command to terminate. If this might cause the
application to miss a signal that would have killed it, then the application should examine the
return value from system() and take whatever action is appropriate to the application if the
command terminated due to receipt of a signal.

The system() function shall not affect the termination status of any child of the calling processes
other than the process or processes it itself creates.

The system() function shall not return until the child process has terminated.

RETURN VALUE
If command is a null pointer, system() shall return non-zero to indicate that a command processor
is available, or zero if none is available. The system() function shall always return non-zero when
command is NULL.

If command is not a null pointer, system() shall return the termination status of the command
language interpreter in the format specified by waitpid(). The termination status shall be as
defined for the sh utility; otherwise, the termination status is unspecified. If some error prevents
the command language interpreter from executing after the child process is created, the return
value from system() shall be as if the command language interpreter had terminated using
exit(127) or _exit(127). If a child process cannot be created, or if the termination status for the
command language interpreter cannot be obtained, system() shall return -1 and set errno to
indicate the error.

ERRORS
The system() function may set errno values as described by fork().

In addition, system() may fail if:

ECHILD The status of the child process created by system() is no longer available.
EXAMPLES

None.

APPLICATION USAGE

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>.

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.

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 (“ ”, 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.

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.

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.

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.

RATIONALE

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 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.

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.

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.
Finally, IEEE Std 1003.1-2001 requires the command to be interpreted as in the shell command language defined in the Shell and Utilities volume of IEEE Std 1003.1-2001.

Note that, `system(NULL)` is required to return non-zero, indicating that there is a command language interpreter. At first glance, this would seem to conflict with the ISO C standard which allows `system(NULL)` to return zero. There is no conflict, however. A system must have a command language interpreter, and is non-conforming if none is present. It is therefore permissible for the `system()` function on such a system to implement the behavior specified by the ISO C standard as long as it is understood that the implementation does not conform to IEEE Std 1003.1-2001 if `system(NULL)` returns zero.

It was explicitly decided that when `command` is NULL, `system()` should not be required to check to make sure that the command language interpreter actually exists with the correct mode, that there are enough processes to execute it, and so on. The call `system(NULL)` could, theoretically, check for such problems as too many existing child processes, and return zero. However, it would be inappropriate to return zero due to such a (presumably) transient condition. If some condition exists that is not under the control of this application and that would cause any `system()` call to fail, that system has been rendered non-conforming.

Early drafts required, or allowed, `system()` to return with `errno` set to [EINTR] if it was interrupted with a signal. This error return was removed, and a requirement that `system()` not return until the child has terminated was added. This means that if a `waitpid()` call in `system()` exits with `errno` set to [EINTR], `system()` must reissue the `waitpid()`. This change was made for two reasons:

1. There is no way for an application to clean up if `system()` returns [EINTR], short of calling `wait()`, and that could have the undesirable effect of returning the status of children other than the one started by `system()`.

2. While it might require a change in some historical implementations, those implementations already have to be changed because they use `wait()` instead of `waitpid()`.

Note that if the application is catching SIGCHLD signals, it will receive such a signal before a successful `system()` call returns.

To conform to IEEE Std 1003.1-2001, `system()` must use `waitpid()`, or some similar function, instead of `wait()`.

The following code sample illustrates how `system()` might be implemented on an implementation conforming to IEEE Std 1003.1-2001.

```c
#include <signal.h>
int system(const char *cmd)
{
  int stat;
  pid_t pid;
  struct sigaction sa, savintr, savequit;
  sigset_t saveblock;
  if (cmd == NULL)
    return(1);
  sigemptyset(&sa.sa_mask);
  sa.sa_handler = SIG_IGN;
  sigemptyset(&savintr.sa_mask);
  sa.sa_flags = 0;
  sigemptyset(&savequit.sa_mask);
  sigaction(SIGINT, &sa, &savintr);
  sigaction(SIGQUIT, &sa, &savequit);
  /* run the command and wait for it to complete */
  pid = fork();
  if (pid == 0)
  {
    execvp(cmd, argv);
    exit(1);
  }
  int status;
  waitpid(pid, &status, 0);
  return (status);
}
```
sigaddset(&sa.sa_mask, SIGCHLD);
sigprocmask(SIG_BLOCK, &sa.sa_mask, &saveblock);
if ((pid = fork()) == 0) {
    sigaction(SIGINT, &savintr, (struct sigaction *)0);
    sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
    sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
    execl(”/bin/sh”, "sh", "-c", cmd, (char *)0);
    _exit(127);
}
if (pid == -1) {
    stat = -1; /* errno comes from fork() */
} else {
    while (waitpid(pid, &stat, 0) == -1) {
        if (errno != EINTR){
            stat = -1;
            break;
        }
    }
    sigaction(SIGINT, &savintr, (struct sigaction *)0);
    sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
    sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
    return(stat);
}

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.

FUTURE DIRECTIONS
None.

SEE ALSO
exec, pipe(), waitpid(), the Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>,
<signal.h>, <stdlib.h>, <sys/wait.h>, the Shell and Utilities volume of IEEE Std 1003.1-2001, sh

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.
NAME
  tan, tanf, tanl — tangent function

SYNOPSIS
  #include <math.h>
  
  double tan(double x);
  float tanf(float x);
  long double tanl(long double x);

DESCRIPTION
  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
  
  These functions shall compute the tangent of their argument \( x \), measured in radians.

  An application wishing to check for error situations should set \( errno \) to zero and call
  \( feclearexcept(\text{FE\_ALL\_EXCEPT}) \) before calling these functions. On return, if \( errno \) is non-zero or
  \( fetestexcept(\text{FE\_INVALID | FE\_DIVBYZERO | FE\_OVERFLOW | FE\_UNDERFLOW}) \) is non-
  zero, an error has occurred.

RETURN VALUE
  Upon successful completion, these functions shall return the tangent of \( x \).

  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.

  If \( x \) is NaN, a NaN shall be returned.

  If \( x \) is ±0, \( x \) shall be returned.

  If \( x \) is subnormal, a range error may occur and \( x \) should be returned.

  If \( x \) is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-
  defined value shall be returned.

  If the correct value would cause underflow, and is representable, a range error may occur and
  the correct value shall be returned.

  If the correct value would cause overflow, a range error shall occur and \( \tan() \), \( \tanf() \), and \( \tanl() \)
  shall return the value of the macro \( \text{HUGE\_VAL, HUGE\_VALF, and HUGE\_VALL} \), respectively.

ERRORS
  These functions shall fail if:

  Domain Error The value of \( x \) is ±Inf.

  If the integer expression (\( \text{math\_erhandling} \& \text{MATH\_ERRNO} \)) is non-zero,
  then \( errno \) shall be set to [\text{EDOM}]. If the integer expression (\( \text{math\_erhandling} \& \text{MATH\_ERREXCEPT} \)) is non-zero, then the invalid floating-point exception
  shall be raised.

  Range Error The result overflows

  If the integer expression (\( \text{math\_erhandling} \& \text{MATH\_ERRNO} \)) is non-zero,
  then \( errno \) shall be set to [\text{ERANGE}]. If the integer expression (\( \text{math\_erhandling} \& \text{MATH\_ERREXCEPT} \)) is non-zero, then the overflow
  floating-point exception shall be raised.

  These functions may fail if:
EXAMPLES

Taking the Tangent of a 45-Degree Angle

```c
#include <math.h>
...
double radians = 45.0 * M_PI / 180;
double result;
...
result = tan (radians);
```

APPLICATION USAGE

There are no known floating-point representations such that for a normal argument, \(\tan(x)\) is either overflow or underflow.

These functions may lose accuracy when their argument is near a multiple of \(\pi/2\) or is far from 0.0.

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

\(\text{atan}()\), \(\text{fpelearexcept}()\), \(\text{fetestexcept}()\), \(\text{isnan}()\), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, \(<\text{math.h}>\)

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

Issue 6

The \(\tanf()\) and \(\tanl()\) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

tanh()

NAME

tanh, tanhf, tanhl — hyperbolic tangent functions

SYNOPSIS

#include <math.h>

double tanh(double x);
floating tanhf(floating x);
long double tanhl(long double x);

DESCRIPTION

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.

These functions shall compute the hyperbolic tangent of their argument x.

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.

RETURN VALUE

Upon successful completion, these functions shall return the hyperbolic tangent of x.

If x is NaN, a NaN shall be returned.
If x is ±0, x shall be returned.
If x is ±Inf, ±1 shall be returned.
If x is subnormal, a range error may occur and x should be returned.

ERRORS

These functions may fail if:

Range Error The value of x is subnormal.

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.

EXAMPLES

None.

APPLICATION USAGE

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.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

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>
CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The tanhf( ) and tanhl( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

NAME
tanl — tangent function

SYNOPSIS
#include <math.h>
long double tanl(long double x);

DESCRIPTION
Refer to tan().
NAME
tcdrain — wait for transmission of output

SYNOPSIS
#include <termios.h>
int tcdrain(int fildes);

DESCRIPTION
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. 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.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and errno set to indicate the error.

ERRORS
The tcdrain() function shall fail if:

[EBADF] The fildes argument is not a valid file descriptor.
[EINTR] A signal interrupted tcdrain().
[ENOTTY] The file associated with fildes is not a terminal.

The tcdrain() function may fail if:

[EIO] The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
tcflush(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 6
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

• In the DESCRIPTION, the final paragraph is no longer conditional on _POSIX_JOB_CONTROL. This is a FIPS requirement.
tcdrain()

- The [EIO] error is added.
NAME
tcflow — suspend or restart the transmission or reception of data

SYNOPSIS
#include <termios.h>

int tcflow(int fildes, int action);

DESCRIPTION
The tcflow() function shall suspend or restart transmission or reception of data on the object
referred to by fildes, depending on the value of action. The fildes argument is an open file
descriptor associated with a terminal.

• If action is TCOFF, output shall be suspended.
• If action is TCOON, suspended output shall be restarted.
• If action is TCIOFF, the system shall transmit a STOP character, which is intended to cause
  the terminal device to stop transmitting data to the system.
• If action is TCION, the system shall transmit a START character, which is intended to cause
  the terminal device to start transmitting data to the system.

The default on the opening of a terminal file is that neither its input nor its output are
suspended.

Attempts to use tcflow() 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.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to
indicate the error.

ERRORS
The tcflow() function shall fail if:

[EBADF] The fildes argument is not a valid file descriptor.
[EINVAL] The action argument is not a supported value.
[ENOTTY] The file associated with fildes is not a terminal.

The tcflow() function may fail if:

[EIO] The process group of the writing process is orphaned, and the writing process
is not ignoring or blocking SIGTTOU.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO

tcsendbreak(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>, <unistd.h>

CHANGE HISTORY

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] error is added.
NAME
tcflush — flush non-transmitted output data, non-read input data, or both

SYNOPSIS
#include <termios.h>
int tcflush(int fildes, int queue_selector);

DESCRIPTION
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:
- If queue_selector is TCIFLUSH, it shall flush data received but not read.
- If queue_selector is TCOFLUSH, it shall flush data written but not transmitted.
- If queue_selector is TCIOFLUSH, it shall flush both data received but not read and data written but not transmitted.

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.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and errno set to indicate the error.

ERRORS
The tcflush() function shall fail if:
- [EBADF] The fildes argument is not a valid file descriptor.
- [EINVAL] The queue_selector argument is not a supported value.
- [ENOTTY] The file associated with fildes is not a terminal.

The tcflush() function may fail if:
- [EIO] The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
tcdrain(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>, <unistd.h>
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

The Open Group Corrigendum U035/1 is applied. In the ERRORS and APPLICATION USAGE sections, references to `tcflow()` are replaced with `tcflush()`.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the final paragraph is no longer conditional on `_POSIX_JOB_CONTROL`. This is a FIPS requirement.
- The [EIO] error is added.
NAME
tcgetattr — get the parameters associated with the terminal

SYNOPSIS
#include <termios.h>
int tcgetattr(int fildes, struct termios *termios_p);

DESCRIPTION
The tcgetattr() function shall get the parameters associated with the terminal referred to by fildes
and store them in the termios structure referenced by termios_p. The fildes argument is an open
file descriptor associated with a terminal.
The termios_p argument is a pointer to a termios structure.
The tcgetattr() operation is allowed from any process.
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).

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to
indicate the error.

ERRORS
The tcgetattr() function shall fail if:
[EBADF] The fildes argument is not a valid file descriptor.
[ENOTTY] The file associated with fildes is not a terminal.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
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.
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.
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.
Note that either specification has a problem. When zero is returned, it implies everything succeeded even if some of the changes were not made. When \(-1\) is returned, it implies everything failed even though some of the changes were made.

Applications that need all of the requested changes made to work properly should follow \texttt{tcsetattr()} with a call to \texttt{tcgetattr()} and compare the appropriate field values.

\textbf{Future Directions}

None.

\textbf{See Also}

\texttt{tcsetattr()}, the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, \texttt{<termios.h>}

\textbf{Change History}

First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

In the DESCRIPTION, the rate returned as the input baud rate shall be the output rate. Previously, the number zero was also allowed but was obsolescent.
NAME
tcgetpgrp — get the foreground process group ID

SYNOPSIS
#include <unistd.h>
pid_t tcgetpgrp(int fildes);

DESCRIPTION
The tcgetpgrp() function shall return the value of the process group ID of the foreground process group associated with the terminal.

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.

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.

RETURN VALUE
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.

ERRORS
The tcgetpgrp() function shall fail if:

[EBADF] The fildes argument is not a valid file descriptor.

[ENOTTY] The calling process does not have a controlling terminal, or the file is not the controlling terminal.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
setsid(), setpgid(), tcsetpgrp(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

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, text previously conditional on support for _POSIX_JOB_CONTROL is now mandatory. This is a FIPS requirement.
NAME
tcgetsid() — get the process group ID for the session leader for the controlling terminal

SYNOPSIS
XSI
#include <termios.h>

pid_t tcgetsid(int fildes);

DESCRIPTION
The tcgetsid() function shall obtain the process group ID of the session for which the terminal specified by fildes is the controlling terminal.

RETURN VALUE
Upon successful completion, tcgetsid() shall return the process group ID associated with the terminal. Otherwise, a value of (pid_t)−1 shall be returned and errno set to indicate the error.

ERRORS
The tcgetsid() function shall fail if:

[EBADF] The fildes argument is not a valid file descriptor.
[ENOTTY] The calling process does not have a controlling terminal, or the file is not the controlling terminal.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <termios.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

The [EACCES] error has been removed from the list of mandatory errors, and the description of [ENOTTY] has been reworded.
tcsendbreak()

NAME
tcsendbreak — send a break for a specific duration

SYNOPSIS
#include <termios.h>

int tcsendbreak(int fildes, int duration);

DESCRIPTION
If the terminal is using asynchronous serial data transmission, tcsendbreak() shall cause
transmission of a continuous stream of zero-valued bits for a specific duration. If duration is 0, it
shall cause transmission of zero-valued bits for at least 0.25 seconds, and not more than 0.5
seconds. If duration is not 0, it shall send zero-valued bits for an implementation-defined period
of time.

The fildes argument is an open file descriptor associated with a terminal.

If the terminal is not using asynchronous serial data transmission, it is implementation-defined
whether tcsendbreak() sends data to generate a break condition or returns without taking any
action.

Attempts to use tcsendbreak() 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.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to
indicate the error.

ERRORS
The tcsendbreak() function shall fail if:

EBADF The fildes argument is not a valid file descriptor.
ENOTTY The file associated with fildes is not a terminal.

The tcsendbreak() function may fail if:

EIO The process group of the writing process is orphaned, and the writing process
is not ignoring or blocking SIGTTOU.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface,
<termios.h>, <unistd.h>
CHANGE HISTORY
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 6
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now mandated. This is a FIPS requirement.
- The [EIO] error is added.
tcsetattr() — set the parameters associated with the terminal

#include <termios.h>

int tcsetattr(int fildes, int optional_actions, const struct termios *termios_p);

The tcsetattr() function shall set the parameters associated with the terminal referred to by the open file descriptor fildes (an open file descriptor associated with a terminal) from the termios structure referenced by termios_p as follows:

- If optional_actions is TCSANOW, the change shall occur immediately.
- If optional_actions is TCSADRAIN, the change shall occur after all output written to fildes is transmitted. This function should be used when changing parameters that affect output.
- If optional_actions is TCSAFLUSH, the change shall occur after all output written to fildes is transmitted, and all input so far received but not read shall be discarded before the change is made.

If the output baud rate stored in the termios structure pointed to by termios_p is the zero baud rate, B0, the modem control lines shall no longer be asserted. Normally, this shall disconnect the line.

If the input baud rate stored in the termios structure pointed to by termios_p is 0, the input baud rate given to the hardware is the same as the output baud rate stored in the termios structure.

The tcsetattr() function shall return successfully if it was able to perform any of the requested actions, even if some of the requested actions could not be performed. It shall set all the attributes that the implementation supports as requested and leave all the attributes not supported by the implementation unchanged. If no part of the request can be honored, it shall return -1 and set errno to [EINVAL]. If the input and output baud rates differ and are a combination that is not supported, neither baud rate shall be changed. A subsequent call to tcgetattr() shall return the actual state of the terminal device (reflecting both the changes made and not made in the previous tcsetattr() call). The tcsetattr() function shall not change the values found in the termios structure under any circumstances.

The effect of tcsetattr() is undefined if the value of the termios structure pointed to by termios_p was not derived from the result of a call to tcgetattr() on fildes; an application should modify only fields and flags defined by this volume of IEEE Std 1003.1-2001 between the call to tcgetattr() and tcsetattr(), leaving all other fields and flags unmodified.

No actions defined by this volume of IEEE Std 1003.1-2001, other than a call to tcsetattr() or a close of the last file descriptor in the system associated with this terminal device, shall cause any of the terminal attributes defined by this volume of IEEE Std 1003.1-2001 to change.

If tcsetattr() is called from a process which is a member of a background process group on a fildes associated with its controlling terminal:

- If the calling process is blocking or ignoring SIGTTOU signals, the operation completes normally and no signal is sent.
- Otherwise, a SIGTTOU signal shall be sent to the process group.
System Interfaces

tcsetattr()

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to indicate the error.

ERRORS
The tcsetattr() function shall fail if:

- [EBADF] The fildes argument is not a valid file descriptor.
- [EINVAL] The optional_actions argument is not a supported value, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.
- [ENOTTY] The file associated with fildes is not a terminal.

ERRORS
The tcsetattr() function may fail if:

- [EIO] The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES
None.

APPLICATION USAGE
If trying to change baud rates, applications should call tcsetattr() then call tcgetattr() in order to determine what baud rates were actually selected.

RATIONALE
The tcsetattr() function can be interrupted in the following situations:

- It is interrupted while waiting for output to drain.
- It is called from a process in a background process group and SIGTTOU is caught.

See also the RATIONALE section in tcgetattr().

FUTURE DIRECTIONS
Using an input baud rate of 0 to set the input rate equal to the output rate may not necessarily be supported in a future version of this volume of IEEE Std 1003.1-2001.

SEE ALSO
cfgetispeed(), tcgetattr(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 11, General Terminal Interface, <termios.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.

Issue 6
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now mandated. This is a FIPS requirement.
- The [EIO] error is added.
In the DESCRIPTION, the text describing use of `tcsetattr()` from a process which is a member of a background process group is clarified.
NAME
tcsetpgrp — set the foreground process group ID

SYNOPSIS
#include <unistd.h>

int tcsetpgrp(int fildes, pid_t pgid_id);

DESCRIPTION
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.

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.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to
indicate the error.

ERRORS
The tcsetpgrp() function shall fail if:

[EBADF] The fildes argument is not a valid file descriptor.
[EINVAL] This implementation does not support the value in the pgid_id argument.
[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.
[EPERM] The value of pgid_id 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.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
tcgetpgrp(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/types.h>, <unistd.h>

CHANGE HISTORY
First released in Issue 3. Included for alignment with the POSIX.1-1988 standard.
In the SYNOPSIS, the inclusion of `<sys/types.h>` is no longer required.

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 and ERRORS sections, text previously conditional on `_POSIX_JOB_CONTROL` is now mandated. This is a FIPS requirement.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The Open Group Corrigendum U047/4 is applied.
NAME
tdelete, tfind, tsearch, twalk — manage a binary search tree

SYNOPSIS
#include <search.h>

void *tdelete(const void *restrict key, void **restrict rootp,
               int (*compar)(const void *, const void *));
void *tfind(const void *key, void *const *rootp,
            int (*compar)(const void *, const void *));
void *tsearch(const void *key, void **rootp,
              int (*compar)(const void *, const void *));
void twalk(const void *root,
           void (*action)(const void *, VISIT, int));

DESCRIPTION
The tdelete(), tfind(), tsearch(), and twalk() functions manipulate binary search trees. Comparisons are made with a user-supplied routine, the address of which is passed as the compar argument. This routine is called with two arguments, which are the pointers to the elements being compared. The application shall ensure that the user-supplied routine returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to, or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The tsearch() function shall build and access the tree. The key argument is a pointer to an element to be accessed or stored. If there is a node in the tree whose element is equal to the value pointed to by key, a pointer to this found node shall be returned. Otherwise, the value pointed to by key shall be inserted (that is, a new node is created and the value of key is copied to this node), and a pointer to this node returned. Only pointers are copied, so the application shall ensure that the calling routine stores the data. The rootp argument points to a variable that points to the root node of the tree. A null pointer value for the variable pointed to by rootp denotes an empty tree; in this case, the variable shall be set to point to the node which shall be at the root of the new tree.

Like tsearch(), tfind() shall search for a node in the tree, returning a pointer to it if found. However, if it is not found, tfind() shall return a null pointer. The arguments for tfind() are the same as for tsearch().

The tdelete() function shall delete a node from a binary search tree. The arguments are the same as for tsearch(). The variable pointed to by rootp shall be changed if the deleted node was the root of the tree. The tdelete() function shall return a pointer to the parent of the deleted node, or a null pointer if the node is not found.

The twalk() function shall traverse a binary search tree. The root argument is a pointer to the root node of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) The argument action is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument shall be the address of the node being visited. The structure pointed to by this argument is unspecified and shall not be modified by the application, but it shall be possible to cast a pointer-to-node into a pointer-to-pointer-to-element to access the element stored in the node. The second argument shall be a value from an enumeration data type:

typedef enum { preorder, postorder, endorder, leaf } VISIT;
tdelte() (defined in `<search.h>`), depending on whether this is the first, second, or third time that the
node is visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a
leaf. The third argument shall be the level of the node in the tree, with the root being level 0.

If the calling function alters the pointer to the root, the result is undefined.

**RETURN VALUE**

If the node is found, both `tsearch()` and `tfind()` shall return a pointer to it. If not, `tfind()` shall
return a null pointer, and `tsearch()` shall return a pointer to the inserted item.

A null pointer shall be returned by `tsearch()` if there is not enough space available to create a new
node.

A null pointer shall be returned by `tdelte()`, `tfind()`, and `tsearch()` if `rootp` is a null pointer on
entry.

The `tdelte()` function shall return a pointer to the parent of the deleted node, or a null pointer if
the node is not found.

The `twalk()` function shall not return a value.

**ERRORS**

No errors are defined.

**EXAMPLES**

The following code reads in strings and stores structures containing a pointer to each string and
a count of its length. It then walks the tree, printing out the stored strings and their lengths in
alphabetical order.

```c
#include <search.h>
#include <string.h>
#include <stdio.h>
#define STRSZ 10000
#define NODSZ 500
struct node { /* Pointers to these are stored in the tree. */
    char  *string;
    int length;
};
char string_space[STRSZ]; /* Space to store strings. */
struct node nodes[NODSZ]; /* Nodes to store. */
void *root = NULL; /* This points to the root. */
int main(int argc, char *argv[])
{
    char  *strptr = string_space;
    struct node  *nodeptr = nodes;
    void print_node(const void *, VISIT, int);
    int i = 0, node_compare(const void *, const void *, const void *);
    while (gets(strptr) != NULL && i++ < NODSZ) {
        /* Set node. */
        nodeptr->string = strptr;
        nodeptr->length = strlen(strptr);
        /* Put node into the tree. */
        (void) tsearch((void *)nodeptr, (void **)&root,
            node_compare);
    }
    return 0;
}
```

---

1502 System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
/* Adjust pointers, so we do not overwrite tree. */
        strptr += nodeptr->length + 1;
        nodeptr++;
    }
    twalk(root, print_node);
    return 0;
}
/* This routine compares two nodes, based on an
   alphabetical ordering of the string field. */
int
node_compare(const void *node1, const void *node2)
{
    return strcmp(((const struct node *) node1)->string,
                   ((const struct node *) node2)->string);
}
/* This routine prints out a node, the second time
   twalk encounters it or if it is a leaf. */
void
print_node(const void *ptr, VISIT order, int level)
{
    const struct node *p = *(const struct node **) ptr;
    if (order == postorder || order == leaf) {
        (void) printf("string = %s, length = %d\n","string = %s, length = %d\n", p->string, p->length);
    }
}

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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
hcreate(), lsearch(), the Base Definitions volume of IEEE Std 1003.1-2001, <search.h>
tdelete()

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

Issue 6

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The restrict keyword is added to the tdelete() prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME
telldir — current location of a named directory stream

SYNOPSIS
XSI
#include <dirent.h>

long telldir(DIR *dirp);

DESCRIPTION
The telldir() function shall obtain the current location associated with the directory stream
specified by dirp.

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().

RETURN VALUE
Upon successful completion, telldir() shall return the current location of the specified directory
stream.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
opendir(), readdir(), seekdir(), the Base Definitions volume of IEEE Std 1003.1-2001, <dirent.h>

CHANGE HISTORY
First released in Issue 2.
tempnam() — create a name for a temporary file

SYNOPSIS

#include <stdio.h>

char *tempnam(const char *dir, const char *pfx);

DESCRIPTION

The tempnam() function shall generate a pathname that may be used for a temporary file. The tempnam() function allows the user to control the choice of a directory. The dir argument points to the name of the directory in which the file is to be created. If dir is a null pointer or points to a string which is not a name for an appropriate directory, the path prefix defined as P_tmpdir in the <stdio.h> header shall be used. If that directory is not accessible, an implementation-defined directory may be used.

Many applications prefer their temporary files to have certain initial letter sequences in their names. The pfx argument should be used for this. This argument may be a null pointer or point to a string of up to five bytes to be used as the beginning of the filename.

Some implementations of tempnam() may use tmpnam() internally. On such implementations, if called more than [TMP_MAX] times in a single process, the behavior is implementation-defined.

RETURN VALUE

Upon successful completion, tempnam() shall allocate space for a string, put the generated pathname in that space, and return a pointer to it. The pointer shall be suitable for use in a subsequent call to free(). Otherwise, it shall return a null pointer and set errno to indicate the error.

ERRORS

The tempnam() function shall fail if:

[ENOMEM] Insufficient storage space is available.

EXAMPLES

Generating a Pathname

The following example generates a pathname for a temporary file in directory /tmp, with the prefix file. After the filename has been created, the call to free() deallocates the space used to store the filename.

#include <stdio.h>
#include <stdlib.h>
...
char *directory = "/tmp";
char *fileprefix = "file";
char *file;
file = tempnam(directory, fileprefix);
free(file);

APPLICATION USAGE

This function only creates pathnames. It is the application’s responsibility to create and remove the files. 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.
RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fopen(), free(), open(), tmpfile(), tmpnam(), unlink(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.
NAME
tfind — search binary search tree

SYNOPSIS
#include <search.h>
void *tfind(const void *key, void *const *rootp,
int (*compar)(const void *, const void *));

DESCRIPTION
Refer to tdelete().
tgamma( )

NAME
tgamma, tgammaf, tgammal — compute gamma() function

SYNOPSIS
#include <math.h>

double tgamma(double x);
float tgammf(float x);
long double tgammal(long double x);

DESCRIPTION
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.

These functions shall compute the gamma() function of x.

An application wishing to check for error situations should set errno to zero and call fecl except(FE_ALL_EXCEPT) before calling these functions. On return, if errno is non-zero or fetest except(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE
Upon successful completion, these functions shall return Gamma(x).

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.

If the correct value would cause overflow, a range error shall occur and tgamma(), tgammaf(), and tgammal() shall return the value of the macro HUGE_VAL, HUGE_VALF, or HUGE_VALL, respectively.

If x is NaN, a NaN shall be returned.

If x is +Inf, x shall be returned.

If x is ±0, a pole error shall occur, and tgamma(), tgammaf(), and tgammal() shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL, respectively.

If x is −Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS
These functions shall fail if:

Domain Error The value of x is a negative integer, or x is −Inf.

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.

Pole Error The value of x is zero.

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.
Range Error The value overflows.

If the integer expression (math_errno & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errno & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

**EXAMPLES**

None.

**APPLICATION USAGE**

For IEEE Std 754-1985 *double*, overflow happens when \(0 < x < 1/\text{DBL\_MAX}\), and \(171.7 < x\).

On error, the expressions (math_errno & MATH_ERRNO) and (math_errno & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

**RATIONALE**

This function is named *tgamma()* in order to avoid conflicts with the historical *gamma()* and *lgamma()* functions.

**FUTURE DIRECTIONS**

It is possible that the error response for a negative integer argument may be changed to a pole error and a return value of ±Inf.

**SEE ALSO**

`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>`

**CHANGE HISTORY**

NAME

time — get time

SYNOPSIS

#include <time.h>

#include <time.h>

SYNOPSIS

time_t time(time_t *tloc);

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any
CX conflict between the requirements described here and the ISO C standard is unintentional. This

CX The time() function shall return the value of time in seconds since the Epoch.

The tloc argument points to an area where the return value is also stored. If tloc is a null pointer,
no value is stored.

RETURN VALUE

Upon successful completion, time() shall return the value of time. Otherwise, (time_t)−1 shall be
returned.

ERRORS

No errors are defined.

EXAMPLES

Getting the Current Time

The following example uses the time() function to calculate the time elapsed, in seconds, since
the Epoch, localtime() to convert that value to a broken-down time, and asctime() to convert the
broken-down time values into a printable string.

#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t result;
    result = time(NULL);
    printf("%s%ju secs since the Epoch\n", 
            asctime(localtime(&result)),
            (uintmax_t)result);
    return(0);
}

This example writes the current time to stdout in a form like this:

835810335 secs since the Epoch
Timing an Event

The following example gets the current time, prints it out in the user's format, and prints the number of minutes to an event being timed.

```c
#include <time.h>
#include <stdio.h>
...

time_t now;
int minutes_to_event;
...
time(&now);
minutes_to_event = ...;
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n",
    minutes_to_event);
...
```

APPLICATION USAGE

None.

RATIONALE

The `time()` function returns a value in seconds (type `time_t`) while `times()` returns a set of values in clock ticks (type `clock_t`). Some historical implementations, such as 4.3 BSD, have mechanisms capable of returning more precise times (see below). A generalized timing scheme to unify these various timing mechanisms has been proposed but not adopted.

Implementations in which `time_t` is a 32-bit signed integer (many historical implementations) fail in the year 2038. IEEE Std 1003.1-2001 does not address this problem. However, the use of the `time_t` type is mandated in order to ease the eventual fix.

The use of the `<time.h>` header instead of `<sys/types.h>` allows compatibility with the ISO C standard.

Many historical implementations (including Version 7) and the 1984 /usr/group standard use `long` instead of `time_t`. This volume of IEEE Std 1003.1-2001 uses the latter type in order to agree with the ISO C standard.

4.3 BSD includes `time()` only as an alternate function to the more flexible `gettimeofday()` function.

FUTURE DIRECTIONS

In a future version of this volume of IEEE Std 1003.1-2001, `time_t` is likely to be required to be capable of representing times far in the future. Whether this will be mandated as a 64-bit type or a requirement that a specific date in the future be representable (for example, 10000 AD) is not yet determined. Systems purchased after the approval of this volume of IEEE Std 1003.1-2001 should be evaluated to determine whether their lifetime will extend past 2038.

SEE ALSO

`asctime()`, `clock()`, `ctime()`, `difftime()`, `gettimeofday()`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `strptime()`, `utime()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<time.h>`

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.
Issue 6

Extensions beyond the ISO C standard are marked.

The EXAMPLES, RATIONALE, and FUTURE DIRECTIONS sections are added.
NAME
timer_create — create a per-process timer (REALTIME)

SYNOPSIS
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clockid, struct sigevent *restrict evp,
        timer_t *restrict timerid);

DESCRIPTION
The timer_create() function shall create a per-process timer using the specified clock, clock_id, as
the timing base. The timer_create() function shall return, in the location referenced by timerid, a
timer ID of type timer_t used to identify the timer in timer requests. This timer ID shall be unique within the calling process until the timer is deleted. The particular clock, clock_id, is defined in <time.h>. The timer whose ID is returned shall be in a disarmed state upon return
from timer_create().

The evp argument, if non-NULL, points to a sigevent structure. This structure, allocated by the application, defines the asynchronous notification to occur as specified in Section 2.4.1 (on page 28) when the timer expires. If the evp argument is NULL, the effect is as if the evp argument pointed to a sigevent structure with the sigev_notify member having the value SIGEV_SIGNAL,
the sigev_signo having a default signal number, and the sigev_value member having the value of the timer ID.

Each implementation shall define a set of clocks that can be used as timing bases for per-process timers. All implementations shall support a clock_id of CLOCK_REALTIME. If the Monotonic Clock option is supported, implementations shall support a clock_id of CLOCK_MONOTONIC.
Per-process timers shall not be inherited by a child process across a fork() and shall be disarmed and deleted by an exec.

If _POSIX_CPUTIME is defined, implementations shall support clock_id values representing the
CPU-time clock of the calling process.

If _POSIX_THREAD_CPUTIME is defined, implementations shall support clock_id values representing the CPU-time clock of the calling thread.

It is implementation-defined whether a timer_create() function will succeed if the value defined
by clock_id corresponds to the CPU-time clock of a process or thread different from the process
or thread invoking the function.

RETURN VALUE
If the call succeeds, timer_create() shall return zero and update the location referenced by timerid
to a timer_t, which can be passed to the per-process timer calls. If an error occurs, the function shall return a value of −1 and set errno to indicate the error. The value of timerid is undefined if an error occurs.

ERRORS
The timer_create() function shall fail if:

[EAGAIN] The system lacks sufficient signal queuing resources to honor the request.

[EAGAIN] The calling process has already created all of the timers it is allowed by this implementation.

[EINVAL] The specified clock ID is not defined.
The implementation does not support the creation of a timer attached to the CPU-time clock that is specified by `clock_id` and associated with a process or thread different from the process or thread invoking `timer_create()`.

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**

**Periodic Timer Overrun and Resource Allocation**

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.

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.

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.

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.

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 functions, as defined, permit an implementation to maintain the overrun count in user space, associated with the `timerid`. The `timer_getoverrun()` function can then be implemented as a macro that uses the `timerid` argument (which may just be a pointer to a user space structure containing the counter) to locate the overrun count with no system call overhead. Other implementations, less concerned with this class of applications, can avoid the asynchronous update of user space by maintaining the count in a system structure at the cost of the extra system call to obtain it.
Timer Expiration Signal Parameters

The Realtime Signals Extension option supports an application-specific datum that is delivered to the extended signal handler. This value is explicitly specified by the application, along with the signal number to be delivered, in a `sigevent` structure. The type of the application-defined value can be either an integer constant or a pointer. This explicit specification of the value, as opposed to always sending the timer ID, was selected based on existing practice.

It is common practice for realtime applications (on non-POSIX systems or realtime extended POSIX systems) to use the parameters of event handlers as the case label of a switch statement or as a pointer to an application-defined data structure. Since `timer_ids` are dynamically allocated by the `timer_create()` function, they can be used for neither of these functions without additional application overhead in the signal handler; for example, to search an array of saved timer IDs to associate the ID with a constant or application data structure.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getres(), timer_delete(), timer_getoverrun(), the Base Definitions volume of IEEE Std 1003.1-2001, `<time.h>`

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The `timer_create()` function is marked as part of the Timers option.

The `[ENOSYS]` error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

CPU-time clocks are added for alignment with IEEE Std 1003.1d-1999.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding the requirement for the `CLOCK_MONOTONIC` clock under the Monotonic Clock option.

The `restrict` keyword is added to the `timer_create()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME

timer_delete — delete a per-process timer (REALTIME)

SYNOPSIS

#include <time.h>

int timer_delete(timer_t timerid);

DESCRIPTION

The timer_delete() function deletes the specified timer, timerid, previously created by the timer_create() function. If the timer is armed when timer_delete() is called, the behavior shall be as if the timer is automatically disarmed before removal. The disposition of pending signals for the deleted timer is unspecified.

RETURN VALUE

If successful, the timer_delete() function shall return a value of zero. Otherwise, the function shall return a value of −1 and set errno to indicate the error.

ERRORS

The timer_delete() function shall fail if:

EINVAL The timer ID specified by timerid is not a valid timer ID.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

timer_create(), the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The timer_delete() function is marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.
NAME
timer_getoverrun, timer_gettime, timer_settime — per-process timers (REALTIME)

SYNOPSIS
#include <time.h>

int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags,
    const struct itimerspec *restrict value,
    struct itimerspec *restrict ovalue);

DESCRIPTION
The timer_gettime() function shall store the amount of time until the specified timer, timerid,
expires and the reload value of the timer into the space pointed to by the value argument. The
it_value member of this structure shall contain the amount of time before the timer expires, or
zero if the timer is disarmed. This value is returned as the interval until timer expiration, even if
the timer was armed with absolute time. The it_interval member of value shall contain the reload
value last set by timer_settime().

The timer_settime() function shall set the time until the next expiration of the timer specified by
timerid from the it_value member of the value argument and arm the timer if the it_value member
of value is non-zero. If the specified timer was already armed when timer_settime() is called, this
call shall reset the time until next expiration to the value specified. If the it_value member of value
is zero, the timer shall be disarmed. The effect of disarming or resetting a timer with pending
expiration notifications is unspecified.

If the flag TIMER_ABSTIME is not set in the argument flags, timer_settime() shall behave as if the
time until next expiration is set to be equal to the interval specified by the it_value member of
value. That is, the timer shall expire in it_value nanoseconds from when the call is made. If the
flag TIMER_ABSTIME is set in the argument flags, timer_settime() shall behave as if the time
until next expiration is set to be equal to the difference between the absolute time specified by
the it_value member of value and the current value of the clock associated with timerid. That is,
the timer shall expire when the clock reaches the value specified by the it_value member of value.
If the specified time has already passed, the function shall succeed and the expiration
notification shall be made.

The reload value of the timer shall be set to the value specified by the it_interval member of
value. When a timer is armed with a non-zero it_interval, a periodic (or repetitive) timer is
specified.

Time values that are between two consecutive non-negative integer multiples of the resolution
of the specified timer shall be rounded up to the larger multiple of the resolution. Quantization
error shall not cause the timer to expire earlier than the rounded time value.

If the argument ovalue is not NULL, the timer_settime() function shall store, in the location
referenced by ovalue, a value representing the previous amount of time before the timer would
have expired, or zero if the timer was disarmed, together with the previous timer reload value.
Timers shall not expire before their scheduled time.

Only a single signal shall be queued to the process for a given timer at any point in time. When a
timer for which a signal is still pending expires, no signal shall be queued, and a timer overrun
shall occur. When a timer expiration signal is delivered to or accepted by a process, if the
implementation supports the Realtime Signals Extension, the timer_getoverrun() function shall
return the timer expiration overrun count for the specified timer. The overrun count returned
contains the number of extra timer expirations that occurred between the time the signal was
generated (queued) and when it was delivered or accepted, up to but not including an
implementation-defined maximum of {DELAYTIMER_MAX}. If the number of such extra
expirations is greater than or equal to {DELAYTIMER_MAX}, then the overrun count shall be set
to {DELAYTIMER_MAX}. The value returned by timer_getoverrun() shall apply to the most
recent expiration signal delivery or acceptance for the timer. If no expiration signal has been
delivered for the timer, or if the Realtime Signals Extension is not supported, the return value of
timer_getoverrun() is unspecified.

RETURN VALUE
If the timer_getoverrun() function succeeds, it shall return the timer expiration overrun count as
explained above.
If the timer_gettime() or timer_settime() functions succeed, a value of 0 shall be returned.
If an error occurs for any of these functions, the value −1 shall be returned, and errno set to
indicate the error.

ERRORS
The timer_getoverrun(), timer_gettime(), and timer_settime() functions shall fail if:
[EINVAL] The timerid argument does not correspond to an ID returned by timer_create()
but not yet deleted by timer_delete().
The timer_settime() function shall fail if:
[EINVAL] A value structure specified a nanosecond value less than zero or greater than
or equal to 1 000 million, and the it_value member of that structure did not
specify zero seconds and nanoseconds.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
Practical clocks tick at a finite rate, with rates of 100 hertz and 1 000 hertz being common. The
inverse of this tick rate is the clock resolution, also called the clock granularity, which in either
case is expressed as a time duration, being 10 milliseconds and 1 millisecond respectively for
these common rates. The granularity of practical clocks implies that if one reads a given clock
twice in rapid succession, one may get the same time value twice; and that timers must wait for
the next clock tick after the theoretical expiration time, to ensure that a timer never returns too
soon. Note also that the granularity of the clock may be significantly coarser than the resolution
of the data format used to set and get time and interval values. Also note that some
implementations may choose to adjust time and/or interval values to exactly match the ticks of
the underlying clock.

This volume of IEEE Std 1003.1-2001 defines functions that allow an application to determine the
implementation-supported resolution for the clocks and requires an implementation to
document the resolution supported for timers and nanosleep() if they differ from the supported
clock resolution. This is more of a procurement issue than a runtime application issue.

FUTURE DIRECTIONS
None.
SEE ALSO

`clock_getres()`, `timer_create()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<time.h>`

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The `timer_getoverrun()`, `timer_gettime()`, and `timer_settime()` functions are marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

The [EINVAL] error condition is updated to include the following: “and the `it_value` member of that structure did not specify zero seconds and nanoseconds.” This change is for IEEE PASC Interpretation 1003.1 #89.

The DESCRIPTION for `timer_getoverrun()` is updated to clarify that “If no expiration signal has been delivered for the timer, or if the Realtime Signals Extension is not supported, the return value of `timer_getoverrun()` is unspecified”.

The `restrict` keyword is added to the `timer_settime()` prototype for alignment with the ISO/IEC 9899:1999 standard.
NAME

times — get process and waited-for child process times

SYNOPSIS

#include <sys/times.h>

clock_t times(struct tms *buffer);

DESCRIPTION

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>.

All times are measured in terms of the number of clock ticks used.

The times of a terminated child process shall be included in the tms_cutime and tms_cstime
elements of the parent when wait() or waitpid() returns the process ID of this terminated child. If
a child process has not waited for its children, their times shall not be included in its times.

- 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.

RETURN VALUE

Upon successful completion, times() shall return the elapsed real time, in clock ticks, since an
arbitrary point in the past (for example, system start-up time). This point does not change from
one invocation of times() within the process to another. The return value may overflow the
possible range of type clock_t. If times() fails, (clock_t)−1 shall be returned and errno set to
indicate the error.

ERRORS

No errors are defined.

EXAMPLES

Timing a Database Lookup

The following example defines two functions, start_clock() and end_clock(), that are used to time
a lookup. It also defines variables of type clock_t and tms to measure the duration of
transactions. The start_clock() function saves the beginning times given by the times() function.
The end_clock() function gets the ending times and prints the difference between the two times.

#include <sys/times.h>
#include <stdio.h>

... void start_clock(void);
void end_clock(char *msg);

... static clock_t st_time;
static clock_t en_time;
static struct tms st_cpu;
static struct tms en_cpu;
...  
void start_clock()
{
    st_time = times(&st_cpu);
}

/* This example assumes that the result of each subtraction
   is within the range of values that can be represented in
   an integer type. */
void end_clock(char *msg)
{
    en_time = times(&en_cpu);
    puts(msg,stdout);
    printf("Real Time: %jd, User Time %jd, System Time %jd\n",
           (intmax_t)(en_time - st_time),
           (intmax_t)(en_cpu.tms_utime - st_cpu.tms_utime),
           (intmax_t)(en_cpu.tms_stime - st_cpu.tms_stime));
}

APPLICATION USAGE
Applications should use \emph{sysconf}(_SC_CLK_TCK) to determine the number of clock ticks per
second as it may vary from system to system.

RATIONALE
The accuracy of the times reported is intentionally left unspecified to allow implementations
flexibility in design, from uniprocessor to multi-processor networks.

The inclusion of times of child processes is recursive, so that a parent process may collect the
total times of all of its descendants. But the times of a child are only added to those of its parent
when its parent successfully waits on the child. Thus, it is not guaranteed that a parent process
can always see the total times of all its descendants; see also the discussion of the term
"\emph{realtime}" in \emph{alarm}().

If the type \texttt{clock_t} is defined to be a signed 32-bit integer, it overflows in somewhat more than a
year if there are 60 clock ticks per second, or less than a year if there are 100. There are individual
systems that run continuously for longer than that. This volume of IEEE Std 1003.1-2001 permits
an implementation to make the reference point for the returned value be the start-up time of the
process, rather than system start-up time.

The term "\emph{charge}" in this context has nothing to do with billing for services. The operating
system accounts for time used in this way. That information must be correct, regardless of how
that information is used.

FUTURE DIRECTIONS
None.

SEE ALSO
\emph{alarm}(), \emph{exec}, \emph{fork}(), \emph{sysconf}(), \emph{time}(), \emph{wait}(), the Base Definitions volume of
IEEE Std 1003.1-2001, \texttt{<sys/times.h>}

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
timezone — difference from UTC and local standard time

SYNOPSIS

```c
#include <time.h>
extern long timezone;
```

DESCRIPTION

Refer to `tzset()`.
**NAME**

tmpfile — create a temporary file

**SYNOPSIS**

```c
#include <stdio.h>

FILE *tmpfile(void);
```

**DESCRIPTION**

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 `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 `fopen()` for update (w+).

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()`.

An error message may be written to standard error if the stream cannot be opened.

**RETURN VALUE**

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.

**ERRORS**

The `tmpfile()` function shall fail if:

- `[EINTR]` A signal was caught during `tmpfile()`.
- `[EMFILE]` [OPEN_MAX] file descriptors are currently open in the calling process.
- `[ENFILE]` The maximum allowable number of files is currently open in the system.
- `[ENOSPC]` The directory or file system which would contain the new file cannot be expanded.
- `[EOVERFLOW]` The file is a regular file and the size of the file cannot be represented correctly in an object of type `off_t`.

The `tmpfile()` function may fail if:

- `[EMFILE]` [FOPEN_MAX] streams are currently open in the calling process.
- `[ENOMEM]` Insufficient storage space is available.

**EXAMPLES**

**Creating a Temporary File**

The following example creates a temporary file for update, and returns a pointer to a stream for the created file in the `fp` variable.

```c
#include <stdio.h>
...
FILE *fp;
fp = tmpfile();
```
APPLICATION USAGE
It should be possible to open at least [TMP_MAX] temporary files during the lifetime of the program (this limit may be shared with tmpnam()) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files ([FOPEN_MAX]).

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fopen(), tmpnam(), unlink(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
Large File Summit extensions are added.
The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

Issue 6
Extensions beyond the ISO C standard are marked.
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:
  • In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.
  • The [EMFILE] optional error condition is added.
The APPLICATION USAGE section is added for alignment with the ISO/IEC 9899:1999 standard.
tmpnam( )

NAME
tmpnam — create a name for a temporary file

SYNOPSIS
#include <stdio.h>
char *tmpnam(char *);

DESCRIPTION
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 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.

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.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-2001 calls tmpnam().

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-NULL parameter.

RETURN VALUE
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.

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.

ERRORS
No errors are defined.

EXAMPLES
Generating a Filename
The following example generates a unique filename and stores it in the array pointed to by ptr.

#include <stdio.h>
...
char filename[L_tmpnam+1];
char *ptr;
ptr = tmpnam(filename);

APPLICATION USAGE
This function only creates filenames. It is the application's responsibility to create and remove the files.

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.
tmpnam()  

**RATIONALE**  
None.  

**FUTURE DIRECTIONS**  
None.  

**SEE ALSO**  
fopen(), open(), tempnam(), tmpfile(), unlink(), the Base Definitions volume of IEEE Std 1003.1-2001, `<stdio.h>`  

**CHANGE HISTORY**  
First released in Issue 1. Derived from Issue 1 of the SVID.  

**Issue 5**  
The DESCRIPTION is updated for alignment with the POSIX Threads Extension.  

**Issue 6**  
Extensions beyond the ISO C standard are marked.  

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.  

The DESCRIPTION is expanded for alignment with the ISO/IEC 9899:1999 standard.
toascii()

NAME
toascii — translate an integer to a 7-bit ASCII character

SYNOPSIS
XSI
#include <ctype.h>

int toascii(int c);

DESCRIPTION
The toascii() function shall convert its argument into a 7-bit ASCII character.

RETURN VALUE
The toascii() function shall return the value (c &0x7f).

ERRORS
No errors are returned.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
isascii(), the Base Definitions volume of IEEE Std 1003.1-2001, <ctype.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
tolower — transliterate uppercase characters to lowercase

SYNOPSIS
#include <ctype.h>
int tolower(int c);

DESCRIPTION
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 tolower() function has as a domain a type int, the value of which is representable as an unsigned char or the value of EOF. If the argument has any other value, the behavior is undefined. If the argument of tolower() represents an uppercase letter, and there exists a corresponding lowercase letter (as defined by character type information in the program locale category LC_CTYPE), the result shall be the corresponding lowercase letter. All other arguments in the domain are returned unchanged.

RETURN VALUE
Upon successful completion, tolower() shall return the lowercase letter corresponding to the argument passed; otherwise, it shall return the argument unchanged.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.
NAME
toupper — transliterate lowercase characters to uppercase

SYNOPSIS
#include <ctype.h>
int toupper(int c);

DESCRIPTION
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

The toupper() function has as a domain a type int, the value of which is representable as an
unsigned char or the value of EOF. If the argument has any other value, the behavior is
undefined. If the argument of toupper() represents a lowercase letter, and there exists a
corresponding uppercase letter (as defined by character type information in the program locale
category LC_CTYPE), the result shall be the corresponding uppercase letter. All other arguments
in the domain are returned unchanged.

RETURN VALUE
Upon successful completion, toupper() shall return the uppercase letter corresponding to the
argument passed.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <ctype.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
Extensions beyond the ISO C standard are marked.
towctrans( )

NAME
towctrans — wide-character transliteration

SYNOPSIS
#include <wctype.h>

wint_t towctrans(wint_t wc, wctrans_t desc);

DESCRIPTION
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.

CX
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 the call to wctrans() that returned the value desc. If the value of desc is invalid (that is, not obtained by a call to wctrans() or desc is invalidated by a subsequent call to setlocale() that has affected category LC_CTYPE), the result is unspecified.

CX
An application wishing to check for error situations should set errno to 0 before calling towctrans(). If errno is non-zero on return, an error has occurred.

RETURN VALUE
If successful, the towctrans() function shall return the mapped value of wc using the mapping described by desc. Otherwise, it shall return wc unchanged.

ERRORS
The towctrans() function may fail if:

CX
EINVAL desc contains an invalid transliteration descriptor.

EXAMPLES
None.

APPLICATION USAGE
The strings "tolower" and "toupper" are reserved for the standard mapping names. In the table below, the functions in the left column are equivalent to the functions in the right column.

towlower(wc) towctrans(wc, wctrans("tolower"))
towupper(wc) towctrans(wc, wctrans("toupper"))

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
towlower(), towupper(), wctrans(), the Base Definitions volume of IEEE Std 1003.1-2001, <wctype.h>

CHANGE HISTORY

Issue 6
Extensions beyond the ISO C standard are marked.
towlower() System Interfaces

NAME
towlower — transliterate uppercase wide-character code to lowercase

SYNOPSIS
#include <wctype.h>
wint_t towlower(wint_t wc);

DESCRIPTION
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 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.

RETURN VALUE
Upon successful completion, towlower() shall return the lowercase letter corresponding to the argument passed; otherwise, it shall return the argument unchanged.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <wctype.h>, <wchar.h>

CHANGE HISTORY
First released in Issue 4.

Issue 5
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>.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
towupper — transliterate lowercase wide-character code to uppercase

SYNOPSIS
#include <wctype.h>
wint_t towupper(wint_t wc);

DESCRIPTION
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 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.

RETURN VALUE
Upon successful completion, towupper() shall return the uppercase letter corresponding to the argument passed. Otherwise, it shall return the argument unchanged.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
setlocale(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 7, Locale, <wctype.h>, <wchar.h>

CHANGE HISTORY
First released in Issue 4.

Issue 5
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>.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
trunc, truncf, truncl — round to truncated integer value

SYNOPSIS
#include <math.h>

double trunc(double x);
float truncf(float x);
long double truncl(long double x);

DESCRIPTION
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.

These functions shall round their argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument.

RETURN VALUE
Upon successful completion, these functions shall return the truncated integer value.

MX If x is NaN, a NaN shall be returned.
If x is ±0 or ±Inf, x shall be returned.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <math.h>

CHANGE HISTORY
**NAME**
truncate — truncate a file to a specified length

**SYNOPSIS**
```
#include <unistd.h>

int truncate(const char *path, off_t length);
```

**DESCRIPTION**
The `truncate()` function shall cause the regular file named by `path` to have a size which shall be equal to `length` bytes.

If the file previously was larger than `length`, the extra data is discarded. If the file was previously shorter than `length`, its size is increased, and the extended area appears as if it were zero-filled.

The application shall ensure that the process has write permission for the file.

If the request would cause the file size to exceed the soft file size limit for the process, the request shall fail and the implementation shall generate the SIGXFSZ signal for the process.

This function shall not modify the file offset for any open file descriptions associated with the file. Upon successful completion, if the file size is changed, this function shall mark for update the `st_ctime` and `st_mtime` fields of the file, and the S_ISUID and S_ISGID bits of the file mode may be cleared.

**RETURN VALUE**
Upon successful completion, `truncate()` shall return 0. Otherwise, −1 shall be returned, and `errno` set to indicate the error.

**ERRORS**
The `truncate()` function shall fail if:

- `[EINTR]` A signal was caught during execution.
- `[EINVAL]` The `length` argument was less than 0.
- `[EFAULT]` or `[EINVAL]` The `length` argument was greater than the maximum file size.
- `[EIO]` An I/O error occurred while reading from or writing to a file system.
- `[EACCES]` A component of the path prefix denies search permission, or write permission is denied on the file.
- `[EISDIR]` The named file is a directory.
- `[ELOOP]` A loop exists in symbolic links encountered during resolution of the `path` argument.
- `[ENAMETOOLONG]` The length of the `path` argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
- `[ENOENT]` A component of `path` does not name an existing file or `path` is an empty string.
- `[ENOTDIR]` A component of the path prefix of `path` is not a directory.
- `[EROFS]` The named file resides on a read-only file system.

The `truncate()` function may fail if:
More than [SYMLOOP_MAX] symbolic links were encountered during resolution of the path argument.

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds [PATH_MAX].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
open(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Moved from X/OPEN UNIX extension to BASE.

Large File Summit extensions are added.

This reference page is split out from the ftruncate() reference page.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.
NAME
    truncf, truncl — round to truncated integer value

SYNOPSIS
    #include <math.h>

        float truncf(float x);
        long double truncl(long double x);

DESCRIPTION
    Refer to trunc().
NAME

tsearch — search a binary search tree

SYNOPSIS

XSI

```c
#include <search.h>

void *tsearch(const void * key, void ** rootp,
               int (*compar)(const void *, const void *));
```

DESCRIPTION

Refer to tdelete().
NAME
ttyname, ttyname_r — find the pathname of a terminal

SYNOPSIS
#include <unistd.h>
char *ttyname(int fildes);

int ttyname_r(int fildes, char *name, size_t namesize);

DESCRIPTION
The ttyname() function shall return a pointer to a string containing a null-terminated pathname of the terminal associated with file descriptor fildes. The return value may point to static data whose content is overwritten by each call.

The ttyname() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The ttyname_r() function shall store the null-terminated pathname of the terminal associated with the file descriptor fildes in the character array referenced by name. The array is namesize characters long and should have space for the name and the terminating null character. The maximum length of the terminal name shall be {TTY_NAME_MAX}.

RETURN VALUE
Upon successful completion, ttyname() shall return a pointer to a string. Otherwise, a null pointer shall be returned and errno set to indicate the error.

If successful, the ttyname_r() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

ERRORS
The ttyname() function may fail if:

[EBADF] The fildes argument is not a valid file descriptor.
[ENOTTY] The fildes argument does not refer to a terminal.

The ttyname_r() function may fail if:

[EBADF] The fildes argument is not a valid file descriptor.
[ENOTTY] The fildes argument does not refer to a terminal.

[ERANGE] The value of namesize is smaller than the length of the string to be returned including the terminating null character.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
The term “terminal” is used instead of the historical term “terminal device” in order to avoid a reference to an undefined term.

The thread-safe version places the terminal name in a user-supplied buffer and returns a non-zero value if it fails. The non-thread-safe version may return the name in a static data area that may be overwritten by each call.
ttname()

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The ttname_r() function is included for alignment with the POSIX Threads Extension.
A note indicating that the ttname() function need not be reentrant is added to the DESCRIPTION.

Issue 6
The ttname_r() function is marked as part of the Thread-Safe Functions option.
The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The statement that errno is set on error is added.
- The [EBADF] and [ENOTTY] optional error conditions are added.
**NAME**

twalk — traverse a binary search tree

**SYNOPSIS**

```c
#include <search.h>

void twalk(const void *root,
           void (*action)(const void *, VISIT, int ));
```

**DESCRIPTION**

Refer to `tdelete()`.
NAME

daylight, timezone, tzname, tzset — set timezone conversion information

SYNOPSIS

#include <time.h>

extern int daylight;
extern long timezone;
extern char *tzname[2];
void tzset(void);

DESCRIPTION

The tzset() function shall use the value of the environment variable TZ to set time conversion information used by ctime(), localtime(), mktime(), and strftime(). If TZ is absent from the environment, implementation-defined default timezone information shall be used.

The tzset() function shall set the external variable tzname as follows:

tzname[0] = "std";
tzname[1] = "dst";

where std and dst are as described in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables.

The tzset() function also shall set the external variable daylight to 0 if Daylight Savings Time conversions should never be applied for the timezone in use; otherwise, non-zero. The external variable timezone shall be set to the difference, in seconds, between Coordinated Universal Time (UTC) and local standard time.

RETURN VALUE

The tzset() function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

Example TZ variables and their timezone differences are given in the table below:

<table>
<thead>
<tr>
<th>TZ</th>
<th>timezone</th>
</tr>
</thead>
<tbody>
<tr>
<td>EST5EDT</td>
<td>5<em>60</em>60</td>
</tr>
<tr>
<td>GMT0</td>
<td>0<em>60</em>60</td>
</tr>
<tr>
<td>JST-9</td>
<td>−9<em>60</em>60</td>
</tr>
<tr>
<td>MET-1MEST</td>
<td>−1<em>60</em>60</td>
</tr>
<tr>
<td>MST7MDT</td>
<td>7<em>60</em>60</td>
</tr>
<tr>
<td>PST8PDT</td>
<td>8<em>60</em>60</td>
</tr>
</tbody>
</table>

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.
SEE ALSO
ctime(), localtime(), mktime(), strftime(), the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables, <time.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
The example is corrected.
NAME
ualarm — set the interval timer

SYNOPSIS
XSI
#include <unistd.h>
useconds_t ualarm(useconds_t useconds, useconds_t interval);

DESCRIPTION
The ualarm() function shall cause the SIGALRM signal to be generated for the calling process
after the number of realtime microseconds specified by the useconds argument has elapsed.
When the interval argument is non-zero, repeated timeout notification occurs with a period in
microseconds specified by the interval argument. If the notification signal, SIGALRM, is not
captured or ignored, the calling process is terminated.
Implementations may place limitations on the granularity of timer values. For each interval
timer, if the requested timer value requires a finer granularity than the implementation supports,
the actual timer value shall be rounded up to the next supported value.
Interactions between ualarm() and any of the following are unspecified:
alarm()
nanosleep()
setitimer()
timer_create()
timer_delete()
timer_getoverrun()
timer_gettime()
timer_settime()
sleep()

RETURN VALUE
The ualarm() function shall return the number of microseconds remaining from the previous
ualarm() call. If no timeouts are pending or if ualarm() has not previously been called, ualarm() shall return 0.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
Applications are recommended to use nanosleep() if the Timers option is supported, or
setitimer(), timer_create(), timer_delete(), timer_getoverrun(), timer_gettime(), or timer_settime() instead of this function.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
alarm(), nanosleep(), setitimer(), sleep(), timer_create(), timer_delete(), timer_getoverrun(), the Base
Definitions volume of IEEE Std 1003.1-2001, <unistd.h>
CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
This function is marked obsolescent.
ulimit()

NAME
ulimit — get and set process limits

SYNOPSIS
#include <ulimit.h>
long ulimit(int cmd, ...);

DESCRIPTION
The ulimit() function shall control process limits. The process limits that can be controlled by this function include the maximum size of a single file that can be written (this is equivalent to using setrlimit() with RLIMIT_FSIZE). The cmd values, defined in <ulimit.h>, include:

- UL_GETFSIZE Return the file size limit (RLIMIT_FSIZE) of the process. The limit shall be in units of 512-byte blocks and shall be inherited by child processes. Files of any size can be read. The return value shall be the integer part of the soft file size limit divided by 512. If the result cannot be represented as a long, the result is unspecified.

- UL_SETFSIZE Set the file size limit for output operations of the process to the value of the second argument, taken as a long, multiplied by 512. If the result would overflow an rlim_t, the actual value set is unspecified. Any process may decrease its own limit, but only a process with appropriate privileges may increase the limit. The return value shall be the integer part of the new file size limit divided by 512.

The ulimit() function shall not change the setting of errno if successful.

As all return values are permissible in a successful situation, an application wishing to check for error situations should set errno to 0, then call ulimit(), and, if it returns -1, check to see if errno is non-zero.

RETURN VALUE
Upon successful completion, ulimit() shall return the value of the requested limit. Otherwise, -1 shall be returned and errno set to indicate the error.

ERRORS
The ulimit() function shall fail and the limit shall be unchanged if:

- EINVAL The cmd argument is not valid.
- EPERM A process not having appropriate privileges attempts to increase its file size limit.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO
getrlimit(), setrlimit(), write(), the Base Definitions volume of IEEE Std 1003.1-2001, <ulimit.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
In the description of UL_SETFSIZE, the text is corrected to refer to rlim_t rather than the spurious rlimit_t.
The DESCRIPTION is updated to indicate that errno is not changed if the function is successful.
NAME
umask — set and get the file mode creation mask

SYNOPSIS
#include <sys/stat.h>

mode_t umask(mode_t cmask);

DESCRIPTION
The umask() function shall set the process’ file mode creation mask to cmask and return the
previous value of the mask. Only the file permission bits of cmask (see <sys/stat.h>) are used; the
meaning of the other bits is implementation-defined.

The process’ file mode creation mask is used during open(), creat(), mkdir(), and mkfifo() to turn
off permission bits in the mode argument supplied. Bit positions that are set in cmask are cleared
in the mode of the created file.

RETURN VALUE
The file permission bits in the value returned by umask() shall be the previous value of the file
mode creation mask. The state of any other bits in that value is unspecified, except that a
subsequent call to umask() with the returned value as cmask shall leave the state of the mask the
same as its state before the first call, including any unspecified use of those bits.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
Unsigned argument and return types for umask() were proposed. The return type and the
argument were both changed to mode_t.

Historical implementations have made use of additional bits in cmask for their implementation-
defined purposes. The addition of the text that the meaning of other bits of the field is
implementation-defined permits these implementations to conform to this volume of

FUTURE DIRECTIONS
None.

SEE ALSO
creat(), mkdir(), mkfifo(), open(), the Base Definitions volume of IEEE Std 1003.1-2001,

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

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.
NAME
uname — get the name of the current system

SYNOPSIS
#include <sys/utsname.h>
int uname(struct utsname *name);

DESCRIPTION
The uname() function shall store information identifying the current system in the structure
pointed to by name.
The uname() function uses the utsname structure defined in <sys/utsname.h>.
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.
The format of each member is implementation-defined.

RETURN VALUE
Upon successful completion, a non-negative value shall be returned. Otherwise, −1 shall be
returned and errno set to indicate the error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
The inclusion of the nodename member in this structure does not imply that it is sufficient
information for interfacing to communications networks.

RATIONALE
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>.
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.
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.
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.

FUTURE DIRECTIONS
None.
SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <sys/utsname.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
ungetc — push byte back into input stream

SYNOPSIS
#include <stdio.h>

int ungetc(int c, FILE *stream);

DESCRIPTION
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 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.

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.

If the value of c equals that of the macro EOF, the operation shall fail and the input stream shall be left unchanged.

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.

RETURN VALUE
Upon successful completion, ungetc() shall return the byte pushed back after conversion. Otherwise, it shall return EOF.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fseek(), getc(), fsetpos(), read(), rewind(), setbuf(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.
NAME
ungetwc — push wide-character code back into the input stream

SYNOPSIS
#include <stdio.h>
#include <wchar.h>
wint_t ungetwc(wint_t wc, FILE *stream);

DESCRIPTION
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 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.

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.

If the value of wc equals that of the macro WEOF, the operation shall fail and the input stream shall be left unchanged.

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.

RETURN VALUE
Upon successful completion, ungetwc() shall return the wide-character code corresponding to the pushed-back character. Otherwise, it shall return WEOF.

ERRORS
The ungetwc() function may fail if:

[EILSEQ] An invalid character sequence is detected, or a wide-character code does not correspond to a valid character.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO

fseek(), fsetpos(), read(), rewind(), setbuf(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>, <wchar.h>

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The Optional Header (OH) marking is removed from <stdio.h>.

Issue 6

The [EILSEQ] optional error condition is marked CX.
NAME
unlink — remove a directory entry

SYNOPSIS
#include <unistd.h>

int unlink(const char *path);

DESCRIPTION
The unlink() function shall remove a link to a file. If path names a symbolic link, unlink() shall
remove the symbolic link named by path and shall not affect any file or directory named by the
contents of the symbolic link. Otherwise, unlink() shall remove the link named by the pathname
pointed to by path and shall decrement the link count of the file referenced by the link.

When the file's link count becomes 0 and no process has the file open, the space occupied by the
file shall be freed and the file shall no longer be accessible. If one or more processes have the file
open when the last link is removed, the link shall be removed before unlink() returns, but the
removal of the file contents shall be postponed until all references to the file are closed.

The path argument shall not name a directory unless the process has appropriate privileges and
the implementation supports using unlink() on directories.

Upon successful completion, unlink() shall mark for update the st_ctime and st_mtime fields of
the parent directory. Also, if the file's link count is not 0, the st_ctime field of the file shall be
marked for update.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno set to
indicate the error. If −1 is returned, the named file shall not be changed.

ERRORS
The unlink() function shall fail and shall not unlink the file if:

[EACCES] Search permission is denied for a component of the path prefix, or write
permission is denied on the directory containing the directory entry to be removed.

[EBUSY] The file named by the path argument cannot be unlinked because it is being
used by the system or another process and the implementation considers this
an error.

[ELOOP] A loop exists in symbolic links encountered during resolution of the path
argument.

[ENAMETOOLONG] The length of the path argument exceeds {PATH_MAX} or a pathname
component is longer than {NAME_MAX}.

[ENOENT] A component of path does not name an existing file or path is an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The file named by path is a directory, and either the calling process does not
have appropriate privileges, or the implementation prohibits using unlink()
on directories.

[EPERM] or [EACCES] The S_ISVTX flag is set on the directory containing the file referred to by the
path argument and the caller is not the file owner, nor is the caller the
directory owner, nor does the caller have appropriate privileges.
The directory entry to be unlinked is part of a read-only file system.

The `unlink()` function may fail and not unlink the file if:

- **EBUSY** The file named by `path` is a named STREAM.
- **ELOOP** More than `SYMLOOP_MAX` symbolic links were encountered during resolution of the `path` argument.
- **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`.
- **ETXTBSY** The entry to be unlinked is the last directory entry to a pure procedure (shared text) file that is being executed.

**EXAMPLES**

**Removing a Link to a File**

The following example shows how to remove a link to a file named `/home/cnd/mod1` by removing the entry named `/modules/pass1`.

```c
#include <unistd.h>
char *path = "/modules/pass1";
int status;
...
status = unlink(path);
```

**Checking for an Error**

The following example fragment creates a temporary password lock file named `LOCKFILE`, which is defined as `/etc/ptmp`, and gets a file descriptor for it. If the file cannot be opened for writing, `unlink()` is used to remove the link between the file descriptor and `LOCKFILE`.

```c
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#define LOCKFILE "/etc/ptmp"

int pfd; /* Integer for file descriptor returned by open call. */
FILE *fpfd; /* File pointer for use in putpwent(). */
...
/* Open password Lock file. If it exists, this is an error. */
if ((pfd = open(LOCKFILE, O_WRONLY| O_CREAT | O_EXCL, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1) {
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
/* Lock file created; proceed with fdopen of lock file so that putpwent() can be used. */
if ((fpfd = fdopen(pfd, "w")) == NULL) {
```
Replacing Files

The following example fragment uses `unlink()` to discard links to files, so that they can be replaced with new versions of the files. The first call removes the link to `LOCKFILE` if an error occurs. Successive calls remove the links to `SAVEFILE` and `PASSWDFILE` so that new links can be created, then removes the link to `LOCKFILE` when it is no longer needed.

```c
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>
#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
#define SAVEFILE "/etc/opasswd"
.../* If no change was made, assume error and leave passwd unchanged. */
if (!valid_change) {
        fprintf(stderr, "Could not change password for user %s
", user);
        unlink(LOCKFILE);
        exit(1);
    }
/* Change permissions on new password file. */
chmod(LOCKFILE, S_IRUSR | S_IRGRP | S_IROTH);
/* Remove saved password file. */
unlink(SAVEFILE);
/* Save current password file. */
link(PASSWDFILE, SAVEFILE);
/* Remove current password file. */
unlink(PASSWDFILE);
/* Save new password file as current password file. */
link(LOCKFILE,PASSWDFILE);
/* Remove lock file. */
unlink(LOCKFILE);
exit(0);
APPLICATION USAGE
Applications should use `rmdir()` to remove a directory.
RATIONALE
Unlinking a directory is restricted to the superuser in many historical implementations for reasons given in `link()` (see also `rename()`).
The meaning of [EBUSY] in historical implementations is “mount point busy”. Since this volume of IEEE Std 1003.1-2001 does not cover the system administration concepts of mounting and unmounting, the description of the error was changed to “resource busy”. (This meaning is used by some device drivers when a second process tries to open an exclusive use device.) The wording is also intended to allow implementations to refuse to remove a directory if it is the root or current working directory of any process.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`close()`, `link()`, `remove()`, `rmdir()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<unistd.h>`

**CHANGE HISTORY**

First released in Issue 1. Derived from Issue 1 of the SVID.

**Issue 5**

The [EBUSY] error is added to the “may fail” part of the ERRORS section.

**Issue 6**

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the effect is specified if `path` specifies a symbolic link.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.
- The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
unlockpt — unlock a pseudo-terminal master/slave pair

SYNOPSIS
#include <stdlib.h>
int unlockpt(int fildes);

DESCRIPTION
The unlockpt() function shall unlock the slave pseudo-terminal device associated with the master to which fildes refers.

Conforming applications shall ensure that they call unlockpt() before opening the slave side of a pseudo-terminal device.

RETURN VALUE
Upon successful completion, unlockpt() shall return 0. Otherwise, it shall return -1 and set errno to indicate the error.

ERRORS
The unlockpt() function may fail if:

[EBADF] The fildes argument is not a file descriptor open for writing.

EINVAL The fildes argument is not associated with a master pseudo-terminal device.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
grantpt(), open(), ptsname(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Moved from X/Open UNIX extension to BASE.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
unsetenv() — remove an environment variable

#include <stdlib.h>

int unsetenv(const char *name);

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.

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.

The unsetenv() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

Upon successful completion, zero shall be returned. Otherwise, −1 shall be returned, errno set to indicate the error, and the environment shall be unchanged.

The unsetenv() function shall fail if:

[EINVAL] The name argument is a null pointer, points to an empty string, or points to a string containing an '=' character.

None.

None.

Refer to the RATIONALE section in setenv().

None.

getenv(), setenv(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>, <sys/types.h>, <unistd.h>

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.
NAME
usleep — suspend execution for an interval

SYNOPSIS
#include <unistd.h>

int usleep(useconds_t useconds);

DESCRIPTION
The usleep() function shall cause the calling thread to be suspended from execution until either
the number of realtime microseconds specified by the argument useconds has elapsed or a signal
is delivered to the calling thread and its action is to invoke a signal-catching function or to
terminate the process. The suspension time may be longer than requested due to the scheduling
of other activity by the system.

The useconds argument shall be less than one million. If the value of useconds is 0, then the call
has no effect.

If a SIGALRM signal is generated for the calling process during execution of usleep() and if the
SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether usleep() returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also
unspecified whether it remains pending after usleep() returns or it is discarded.

If a SIGALRM signal is generated for the calling process during execution of usleep(), except as a
result of a prior call to alarm(), and if the SIGALRM signal is not being ignored or blocked from
delivery, it is unspecified whether that signal has any effect other than causing usleep() to return.

If a signal-catching function interrupts usleep() and examines or changes either the time a
SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or
whether the SIGALRM signal is blocked from delivery, the results are unspecified.

If a signal-catching function interrupts usleep() and calls siglongjmp() or longjmp() to restore an
environment saved prior to the usleep() call, the action associated with the SIGALRM signal and
the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also
unspecified whether the SIGALRM signal is blocked, unless the process’ signal mask is restored
as part of the environment.

Implementations may place limitations on the granularity of timer values. For each interval
timer, if the requested timer value requires a finer granularity than the implementation supports,
the actual timer value shall be rounded up to the next supported value.

Interactions between usleep() and any of the following are unspecified:

nanosleep()
seltimer()
timer_create()
timer_delete()
timer_getoverrun()
timer_settime()
timer_settime()
ularm()
sleep()
RETURN VALUE
Upon successful completion, `usleep()` shall return 0; otherwise, it shall return –1 and set `errno` to indicate the error.

ERRORS
The `usleep()` function may fail if:

- [EINVAL] The time interval specified one million or more microseconds.

EXAMPLES
None.

APPLICATION USAGE
Applications are recommended to use `nanosleep()` if the Timers option is supported, or `setitimer()`, `timer_create()`, `timer_delete()`, `timer_getoverrun()`, `timer_gettime()`, or `timer_settime()` instead of this function.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
`alarm()`, `getitimer()`, `nanosleep()`, `sigaction()`, `sleep()`, `timer_create()`, `timer_delete()`, `timer_getoverrun()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<unistd.h>`

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is changed to indicate that timers are now thread-based rather than process-based.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

This function is marked obsolescent.
NAME
utime — set file access and modification times

SYNOPSIS
#include <utime.h>

int utime(const char *path, const struct utimbuf *times);

DESCRIPTION
The utime() function shall set the access and modification times of the file named by the path argument.

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.

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.

The utimbuf structure is defined in the <utime.h> header. The times in the structure utimbuf are measured in seconds since the Epoch.

Upon successful completion, utime() shall mark the time of the last file status change, st_ctime, to be updated; see <sys/stat.h>.

RETURN VALUE
Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno shall be set to indicate the error, and the file times shall not be affected.

ERRORS
The utime() function shall fail if:

[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.

[ELOOP] A loop exists in symbolic links encountered during resolution of the path argument.

[ENAMETOOLONG] The length of the path argument exceeds [PATH_MAX] or a pathname component is longer than [NAME_MAX].

[ENOENT] A component of path does not name an existing file or path is an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

[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.

[EROFS] The file system containing the file is read-only.

The utime() function may fail if:

[ELOOP] More than [SYMLOOP_MAX] symbolic links were encountered during resolution of the path argument.
utime()

[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].

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
The actime structure member must be present so that an application may set it, even though an implementation may ignore it and not change the access time on the file. If an application intends to leave one of the times of a file unchanged while changing the other, it should use stat() to retrieve the file's st_atime and st_mtime parameters, set actime and modtime in the buffer, and change one of them before making the utime() call.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <sys/stat.h>, <utime.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6
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.

• The [ELOOP] mandatory error condition is added.

• A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

• The [ELOOP] optional error condition is added.

• The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
utimes() — set file access and modification times (LEGACY)

#include <sys/time.h>

int utimes(const char *path, const struct timeval times[2]);

DESCRIPTION

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.

For utimes(), the times argument is an array of timeval structures. The first array member represents the date and time of last access, and the second member represents the date and time of last modification. The times in the timeval structure are measured in seconds and microseconds since the Epoch, although rounding toward the nearest second may occur.

If the times argument 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 has write access to the file or appropriate privileges to use this call in this manner. Upon completion, utimes() shall mark the time of the last file status change, st_ctime, for update.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and errno shall be set to indicate the error, and the file times shall not be affected.

ERRORS

The utimes() function shall fail if:

[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 and write access is denied.

[ELOOP] A loop exists in symbolic links encountered during resolution of the path argument.

[ENAMETOOLONG] The length of the path argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

[ENOENT] A component of path does not name an existing file or path is an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The times argument is not a null pointer and the calling process' effective user ID has write access to the file but does not match the owner of the file and the calling process does not have the appropriate privileges.

[EROFS] The file system containing the file is read-only.

The utimes() function may fail if:

[ELOOP] More than SYMLOOP_MAX symbolic links were encountered during resolution of the path argument.

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
EXAMPLES
None.

APPLICATION USAGE
For applications portability, the \texttt{utime()} function should be used to set file access and modification times instead of \texttt{utimes()}.

RATIONALE
None.

FUTURE DIRECTIONS
This function may be withdrawn in a future version.

SEE ALSO
\texttt{utime()}, the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<sys/time.h>}

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
This function is marked LEGACY.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The wording of the mandatory \texttt{[ELOOP]} error condition is updated, and a second optional \texttt{[ELOOP]} error condition is added.
NAME
va_arg, va_copy, va_end, va_start — handle variable argument list

SYNOPSIS
#include <stdarg.h>

type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, argN);

DESCRIPTION
Refer to the Base Definitions volume of IEEE Std 1003.1-2001, <stdarg.h>.
NAME
vfork — create a new process; share virtual memory

SYNOPSIS
#include <unistd.h>

pid_t vfork(void);

DESCRIPTION
The vfork() function shall be equivalent to fork(), except that the behavior is undefined
if the process created by vfork() either modifies any data other than a variable of type pid_t used
to store the return value from vfork(), or returns from the function in which vfork() was called, or
calls any other function before successfully calling _exit() or one of the exec family of functions.

RETURN VALUE
Upon successful completion, vfork() shall return 0 to the child process and return the process ID
of the child process to the parent process. Otherwise, -1 shall be returned to the parent, no child
process shall be created, and errno shall be set to indicate the error.

ERRORS
The vfork() function shall fail if:

[EAGAIN] The system-wide limit on the total number of processes under execution
would be exceeded, or the system-imposed limit on the total number of
processes under execution by a single user would be exceeded.

[ENOMEM] There is insufficient swap space for the new process.

EXAMPLES
None.

APPLICATION USAGE
Conforming applications are recommended not to depend on vfork(), but to use fork() instead.
The vfork() function may be withdrawn in a future version.
On some implementations, vfork() is equivalent to fork().
The vfork() function differs from fork() only in that the child process can share code and data
with the calling process (parent process). This speeds cloning activity significantly at a risk to
the integrity of the parent process if vfork() is misused.
The use of vfork() for any purpose except as a prelude to an immediate call to a function from
the exec family, or to _exit(), is not advised.
The vfork() function can be used to create new processes without fully copying the address
space of the old process. If a forked process is simply going to call exec, the data space copied
from the parent to the child by fork() is not used. This is particularly inefficient in a paged
environment, making vfork() particularly useful. Depending upon the size of the parent’s data
space, vfork() can give a significant performance improvement over fork().
The vfork() function can normally be used just like fork(). It does not work, however, to return
while running in the child’s context from the caller of vfork() since the eventual return from
vfork() would then return to a no longer existent stack frame. Care should be taken, also, to call
_exit() rather than exit() if exec cannot be used, since exit() flushes and closes standard I/O
channels, thereby damaging the parent process’ standard I/O data structures. (Even with fork(),
it is wrong to call exit(), since buffered data would then be flushed twice.)

If signal handlers are invoked in the child process after vfork(), they must follow the same rules
as other code in the child process.
RATIONALE
None.

FUTURE DIRECTIONS
This function may be withdrawn in a future version.

SEE ALSO
exec, exit(), fork(), wait(), the Base Definitions volume of IEEE Std 1003.1-2001, <unistd.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
This function is marked obsolescent.
NAME
vfprintf, vprintf, vsnprintf, vsprintf — format output of a stdarg argument list

SYNOPSIS
#include <stdarg.h>
#include <stdio.h>

int vfprintf(FILE *restrict stream, const char *restrict format,
             va_list ap);
int vprintf(const char *restrict format, va_list ap);
int vsnprintf(char *restrict s, size_t n, const char *restrict format,
              va_list ap);
int vsprintf(char *restrict s, const char *restrict format, va_list ap);

DESCRIPTION
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
The vprintf(), vfprintf(), vsnprintf(), and vsprintf() functions shall be equivalent to printf(),
fprintf(), snprintf(), and sprintf() 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>.
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.

RETURN VALUE
Refer to fprintf().

ERRORS
Refer to fprintf().

EXAMPLES
None.

APPLICATION USAGE
Applications using these functions should call va_end(ap) afterwards to clean up.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fprintf(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdarg.h>, <stdio.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The vsnprintf() function is added.

Issue 6
The vprintf(), vfprintf(), vsnprintf(), and vsprintf() functions are updated for alignment with the
NAME
vfscanf, vscanf, vsscanf — format input of a stdarg argument list

SYNOPSIS
#include <stdarg.h>
#include <stdio.h>

int vfscanf(FILE *restrict stream, const char *restrict format,
             va_list arg);
int vscanf(const char *restrict format, va_list arg);
int vsscanf(const char *restrict s, const char *restrict format,
             va_list arg);

DESCRIPTION
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

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.

RETURN VALUE
Refer to fscanf().

ERRORS
Refer to fscanf().

EXAMPLES
None.

APPLICATION USAGE
Applications using these functions should call va_end(ap) afterwards to clean up.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fscanf(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdarg.h>, <stdio.h>

CHANGE HISTORY
vfwprintf()  

**NAME**  
vfwprintf, vswprintf, vwprintf — wide-character formatted output of a stdarg argument list

**SYNOPSIS**  
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwprintf(FILE *restrict stream, const wchar_t *restrict format, va_list arg);
int vswprintf(wchar_t *restrict ws, size_t n, const wchar_t *restrict format, va_list arg);
int vwprintf(const wchar_t *restrict format, va_list arg);

**DESCRIPTION**  
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.

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>.

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.

**RETURN VALUE**  
Refer to *fwprintf()*.

**ERRORS**  
Refer to *fwprintf()*.

**EXAMPLES**  
None.

**APPLICATION USAGE**  
Applications using these functions should call *va_end(ap)* afterwards to clean up.

**RATIONALE**  
None.

**FUTURE DIRECTIONS**  
None.

**SEE ALSO**  
/fwprintf/(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdarg.h>, <stdio.h>, <wchar.h>

**CHANGE HISTORY**  

**Issue 6**  
The *vfwprintf()* , *vswprintf()* , and *vwprintf()* prototypes are updated for alignment with the ISO/IEC 9899:1999 standard. ()
NAME
vfwscanf, vswscanf, vwscanf — wide-character formatted input of a stdarg argument list

SYNOPSIS
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwscanf(FILE *restrict stream, const wchar_t *restrict format,
va_list arg);
int vswscanf(const wchar_t *restrict ws, const wchar_t *restrict format,
va_list arg);
int vwscanf(const wchar_t *restrict format, va_list arg);

DESCRIPTION
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
The vfwscanf(), vswscanf(), and vwscanf() functions shall be equivalent to the fwscanf(),
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.

RETURN VALUE
Refer to fwscanf().

ERRORS
Refer to fwscanf().

EXAMPLES
None.

APPLICATION USAGE
Applications using these functions should call va_end(ap) afterwards to clean up.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
fwscanf(), the Base Definitions volume of IEEE Std 1003.1-2001, <stdarg.h>, <stdio.h>,
<wchar.h>

CHANGE HISTORY
vprintf() — format the output of a stdarg argument list

SYNOPSIS
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *restrict format, va_list ap);

DESCRIPTION
Refer to vfprintf().
NAME
vscanf — format input of a stdarg argument list

SYNOPSIS
#include <stdarg.h>
#include <stdio.h>

int vscanf(const char *restrict format, va_list arg);

DESCRIPTION
Refer to vfscanf().
vsnprintf(), vsprintf — format output of a stdarg argument list

NAME

vsnprintf, vsprintf — format output of a stdarg argument list

SYNOPSIS

#include <stdarg.h>
#include <stdio.h>

int vsnprintf(char *restrict s, size_t n,
              const char *restrict format, va_list ap);
int vsprintf(char *restrict s, const char *restrict format,
             va_list ap);

DESCRIPTION

Refer to vfprintf().
vsscanf()

NAME
vsscanf — format input of a stdarg argument list

SYNOPSIS
#include <stdarg.h>
#include <stdio.h>

int vsscanf(const char *restrict s, const char *restrict format,
             va_list arg);

DESCRIPTION
Refer to vfscanf().
NAME
vswprintf — wide-character formatted output of a stdarg argument list

SYNOPSIS
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vswprintf(wchar_t *restrict ws, size_t n,
              const wchar_t *restrict format, va_list arg);

DESCRIPTION
Refer to vfwprintf().
NAME
vswscanf — wide-character formatted input of a stdarg argument list

SYNOPSIS
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vswscanf(const wchar_t *restrict ws, const wchar_t *restrict format,
              va_list arg);

DESCRIPTION
Refer to vfscanf().
NAME
vwprintf — wide-character formatted output of a stdarg argument list

SYNOPSIS
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vwprintf(const wchar_t *restrict format, va_list arg);

DESCRIPTION
Refer to vfwprintf().
NAME
vwscanf — wide-character formatted input of a stdarg argument list

SYNOPSIS
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vwscanf(const wchar_t *restrict format, va_list arg);

DESCRIPTION
Refer to vfwscanf().
wait(), System Interfaces

NAME

wait, waitpid — wait for a child process to stop or terminate

SYNOPSIS

#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);

DESCRIPTION

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.

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.

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.

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:

- If pid is equal to (pid_t)−1, status is requested for any child process. In this respect, waitpid() is then equivalent to wait().
- If pid is greater than 0, it specifies the process ID of a single child process for which status is requested.
- If pid is 0, status is requested for any child process whose process group ID is equal to that of the calling process.
- 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.

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:

- WCONTINUED The waitpid() function shall report the status of any continued child process specified by pid whose status has not been reported since it continued from a job control stop.
- WNOHANG The waitpid() function shall not suspend execution of the calling thread if status is not immediately available for one of the child processes specified by pid.
- WUNTRACED The status of any child processes specified by pid that are stopped, and whose status has not yet been reported since they stopped, shall also be reported to the requesting process.

If the calling process has SA_NOCLEWDWAIT 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 wait() and waitpid() shall fail and set errno to [ECHILD].
If `wait()` or `waitpid()` return because the status of a child process is available, these functions shall return a value equal to the process ID of the child process. In this case, if the value of the argument `stat_loc` is not a null pointer, information shall be stored in the location pointed to by `stat_loc`. The value stored at the location pointed to by `stat_loc` shall be 0 if and only if the status returned is from a terminated child process that terminated by one of the following means:

1. The process returned 0 from `main()`.
2. The process called `_exit()` or `exit()` with a `status` argument of 0.
3. The process was terminated because the last thread in the process terminated.

Regardless of its value, this information may be interpreted using the following macros, which are defined in `<sys/wait.h>` and evaluate to integral expressions; the `stat_val` argument is the integer value pointed to by `stat_loc`.

- **WIFEXITED(stat_val)**
  Evaluates to a non-zero value if `status` was returned for a child process that terminated normally.

- **WEXITSTATUS(stat_val)**
  If the value of WIFEXITED(stat_val) is non-zero, this macro evaluates to the low-order 8 bits of the `status` argument that the child process passed to `_exit()` or `exit()`, or the value the child process returned from `main()`.

- **WIFSIGNALED(stat_val)**
  Evaluates to a non-zero value if `status` was returned for a child process that terminated due to the receipt of a signal that was not caught (see `<signal.h>`).

- **WTERMSIG(stat_val)**
  If the value of WIFSIGNALED(stat_val) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

- **WIFSTOPPED(stat_val)**
  Evaluates to a non-zero value if `status` was returned for a child process that is currently stopped.

- **WSTOPSIG(stat_val)**
  If the value of WIFSTOPPED(stat_val) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

- **WIFCONTINUED(stat_val)**
  Evaluates to a non-zero value if `status` was returned for a child process that has continued from a job control stop.

It is unspecified whether the `status` value returned by calls to `wait()` or `waitpid()` for processes created by `posix_spawn()` or `posix_spawnp()` can indicate a WIFSTOPPED(stat_val) before subsequent calls to `wait()` or `waitpid()` indicate WIFEXITED(stat_val) as the result of an error detected before the new process image starts executing.

It is unspecified whether the `status` value returned by calls to `wait()` or `waitpid()` for processes created by `posix_spawn()` or `posix_spawnp()` can indicate a WIFSIGNALED(stat_val) if a signal is sent to the parent’s process group after `posix_spawn()` or `posix_spawnp()` is called.

If the information pointed to by `stat_loc` was stored by a call to `waitpid()` that specified the WUNTRACED flag and did not specify the WCONTINUED flag, exactly one of the macros WIFEXITED(*stat_loc), WIFSIGNALED(*stat_loc), and WIFSTOPPED(*stat_loc) shall evaluate to a non-zero value.
If the information pointed to by `stat_loc` was stored by a call to `waitpid`() that specified the WUNTRACED and WCONTINUED flags, exactly one of the macros WIFEXITED(*`stat_loc`), WIFSIGNALED(*`stat_loc`), WIFSTOPPED(*`stat_loc`), and WIFCONTINUED(*`stat_loc`) shall evaluate to a non-zero value.

If the information pointed to by `stat_loc` was stored by a call to `waitpid`() that did not specify the WUNTRACED or WCONTINUED flags, or by a call to the `wait()` function, exactly one of the macros WIFEXITED(*`stat_loc`), WIFSIGNALED(*`stat_loc`), and WIFCONTINUED(*`stat_loc`) shall evaluate to a non-zero value.

If _POSIX_REALTIME_SIGNALS is defined, and the implementation queues the SIGCHLD signal, then if `wait()` or `waitpid()` returns because the status of a child process is available, any pending SIGCHLD signal associated with the process ID of the child process shall be discarded.

Any other pending SIGCHLD signals shall remain pending.

Otherwise, if SIGCHLD is blocked, if `wait()` or `waitpid()` return because the status of a child process is available, any pending SIGCHLD signal shall be cleared unless the status of another child process is available.

For all other conditions, it is unspecified whether child `status` will be available when a SIGCHLD signal is delivered.

There may be additional implementation-defined circumstances under which `wait()` or `waitpid()` report `status`. This shall not occur unless the calling process or one of its child processes explicitly makes use of a non-standard extension. In these cases the interpretation of the reported `status` is implementation-defined.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes shall be assigned a new parent process ID corresponding to an implementation-defined system process.

**RETURN VALUE**

If `wait()` or `waitpid()` returns because the status of a child process is available, these functions shall return a value equal to the process ID of the child process for which `status` is reported. If `wait()` or `waitpid()` returns due to the delivery of a signal to the calling process, −1 shall be returned and `errno` set to [EINTR]. If `waitpid()` was invoked with WNOHANG set in `options`, it has at least one child process specified by `pid` for which `status` is not available, and `status` is not available for any process specified by `pid`, 0 is returned. Otherwise, (pid_t)−1 shall be returned, and `errno` set to indicate the error.

**ERRORS**

The `wait()` function shall fail if:

- [ECHILD] The calling process has no existing unwaited-for child processes.
- [EINTR] The function was interrupted by a signal. The value of the location pointed to by `stat_loc` is undefined.

The `waitpid()` function shall fail if:

- [ECHILD] The process specified by `pid` does not exist or is not a child of the calling process, or the process group specified by `pid` does not exist or does not have any member process that is a child of the calling process.
wait()

The function was interrupted by a signal. The value of the location pointed to by stat_loc is undefined.

The options argument is not valid.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
A call to the wait() or waitpid() function only returns status on an immediate child process of the calling process; that is, a child that was produced by a single fork() call (perhaps followed by an exec or other function calls) from the parent. If a child produces grandchildren by further use of fork(), none of those grandchildren nor any of their descendants affect the behavior of a wait() from the original parent process. Nothing in this volume of IEEE Std 1003.1-2001 prevents an implementation from providing extensions that permit a process to get status from a grandchild or any other process, but a process that does not use such extensions must be guaranteed to see status from only its direct children.

The waitpid() function is provided for three reasons:

1. To support job control
2. To permit a non-blocking version of the wait() function
3. To permit a library routine, such as system() or pclose(), to wait for its children without interfering with other terminated children for which the process has not waited

The first two of these facilities are based on the wait3() function provided by 4.3 BSD. The function uses the options argument, which is equivalent to an argument to wait3(). The WUNTRACED flag is used only in conjunction with job control on systems supporting job control. Its name comes from 4.3 BSD and refers to the fact that there are two types of stopped processes in that implementation: processes being traced via the ptrace() debugging facility and (untraced) processes stopped by job control signals. Since ptrace() is not part of this volume of IEEE Std 1003.1-2001, only the second type is relevant. The name WUNTRACED was retained because its usage is the same, even though the name is not intuitively meaningful in this context.

The third reason for the waitpid() function is to permit independent sections of a process to spawn and wait for children without interfering with each other. For example, the following problem occurs in developing a portable shell, or command interpreter:

```c
stream = popen("/bin/true");
(void) system("sleep 100");
(void) pclose(stream);
```

On all historical implementations, the final pclose() fails to reap the wait() status of the popen().

The status values are retrieved by macros, rather than given as specific bit encodings as they are in most historical implementations (and thus expected by existing programs). This was necessary to eliminate a limitation on the number of signals an implementation can support that was inherent in the traditional encodings. This volume of IEEE Std 1003.1-2001 does require that a status value of zero corresponds to a process calling _exit(0), as this is the most common encoding expected by existing programs. Some of the macro names were adopted from 4.3 BSD.

These macros syntactically operate on an arbitrary integer value. The behavior is undefined unless that value is one stored by a successful call to wait() or waitpid() in the location pointed to by the stat_loc argument. An early proposal attempted to make this clearer by specifying each
The extension that affects \texttt{wait()} and \texttt{waitpid()} and is common in historical implementations is the \texttt{ptrace()} function. It is called by a child process and causes that child to stop and return a status that appears identical to the status indicated by WIFSTOPPED. The status of \texttt{ptrace()} children is traditionally returned regardless of the WUNTRACED flag (or by the \texttt{wait()} function). Most applications do not need to concern themselves with such extensions because they have control over what extensions they or their children use. However, applications, such as command interpreters, that invoke arbitrary processes may see this behavior when those arbitrary processes misuse such extensions.

Implementations that support \texttt{core} file creation or other implementation-defined actions on termination of some processes traditionally provide a bit in the \texttt{status} returned by \texttt{wait()} to indicate that such actions have occurred.

Allowing the \texttt{wait()} family of functions to discard a pending SIGCHLD signal that is associated with a successfully waited-for child process puts them into the \texttt{sigwait()} and \texttt{sigwaitinfo()} category with respect to SIGCHLD.

This definition allows implementations to treat a pending SIGCHLD signal as accepted by the process in \texttt{wait()}, with the same meaning of ‘accepted’ as when that word is applied to the \texttt{sigwait()} family of functions.

Allowing the \texttt{wait()} family of functions to behave this way permits an implementation to be able to deal precisely with SIGCHLD signals.

In particular, an implementation that does accept (discard) the SIGCHLD signal can make the following guarantees regardless of the queuing depth of signals in general (the list of waitable children can hold the SIGCHLD queue):

1. If a SIGCHLD signal handler is established via \texttt{sigaction()} without the SA_RESETHAND flag, SIGCHLD signals can be accurately counted; that is, exactly one SIGCHLD signal will be delivered to or accepted by the process for every child process that terminates.
2. A single \texttt{wait()} issued from a SIGCHLD signal handler can be guaranteed to return immediately with status information for a child process.
3. When SA_SIGINFO is requested, the SIGCHLD signal handler can be guaranteed to receive a non-NULL pointer to a \texttt{siginfo_t} structure that describes a child process for which a wait via \texttt{waitpid()} or \texttt{waitid()} will not block or fail.
4. The \texttt{system()} function will not cause a process’ SIGCHLD handler to be called as a result of the \texttt{fork()/exec} executed within \texttt{system()} because \texttt{system()} will accept the SIGCHLD signal when it performs a \texttt{waitpid()} for its child process. This is a desirable behavior of \texttt{system()} so that it can be used in a library without causing side effects to the application linked with the library.

An implementation that does not permit the \texttt{wait()} family of functions to accept (discard) a pending SIGCHLD signal associated with a successfully waited-for child, cannot make the guarantees described above for the following reasons:

Guarantee #1

Although it might be assumed that reliable queuing of all SIGCHLD signals generated by the system can make this guarantee, the counter-example is the case of a process that blocks SIGCHLD and performs an indefinite loop of \texttt{fork()/wait()} operations. If the
implementation supports queued signals, then eventually the system will run out of memory for the queue. The guarantee cannot be made because there must be some limit to the depth of queuing.

Guarantees #2 and #3

These cannot be guaranteed unless the wait() family of functions accepts the SIGCHLD signal. Otherwise, a fork()/wait() executed while SIGCHLD is blocked (as in the system() function) will result in an invocation of the handler when SIGCHLD is unblocked, after the process has disappeared.

Guarantee #4

Although possible to make this guarantee, system() would have to set the SIGCHLD handler to SIG_DFL so that the SIGCHLD signal generated by its fork() would be discarded (the SIGCHLD default action is to be ignored), then restore it to its previous setting. This would have the undesirable side effect of discarding all SIGCHLD signals pending to the process.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, exit(), fork(), waitid(), the Base Definitions volume of IEEE Std 1003.1-2001, <signal.h>, <sys/wait.h>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

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.

The following changes were made to align with the IEEE P1003.1a draft standard:

• The processing of the SIGCHLD signal and the [ECHILD] error is clarified.

The semantics of WIFSTOPPED(stat_val), WIFEXITED(stat_val), and WIFSIGNALED(stat_val) are defined with respect to posix_spawn() or posix_spawnp() for alignment with IEEE Std 1003.1d-1999.

The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.
waitid()

NAME
waitid — wait for a child process to change state

SYNOPSIS
XSI
#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

DESCRIPTION
The waitid() function shall suspend the calling thread until one child of the process containing
the calling thread changes state. It records the current state of a child in the structure pointed to
by infop. If a child process changed state prior to the call to waitid(), waitid() shall return
immediately. If more than one thread is suspended in wait() or waitpid() waiting for termination
of the same process, exactly one thread shall return the process status at the time of the target
process termination.

The idtype and id arguments are used to specify which children waitid() waits for.

If idtype is P_PID, waitid() shall wait for the child with a process ID equal to (pid_t)id.
If idtype is P_PGID, waitid() shall wait for any child with a process group ID equal to (pid_t)id.
If idtype is P_ALL, waitid() shall wait for any children and id is ignored.

The options argument is used to specify which state changes waitid() shall wait for. It is formed
by OR’ing together one or more of the following flags:

WEXITED Wait for processes that have exited.
WSTOPPED Status shall be returned for any child that has stopped upon receipt of a signal.
WCONTINUED Status shall be returned for any child that was stopped and has been
continued.
WNOHANG Return immediately if there are no children to wait for.
WNOWAIT Keep the process whose status is returned in infop in a waitable state. This
shall not affect the state of the process; the process may be waited for again
after this call completes.

The application shall ensure that the infop argument points to a siginfo_t structure. If waitid()
returns because a child process was found that satisfied the conditions indicated by the
arguments idtype and options, then the structure pointed to by infop shall be filled in by the
system with the status of the process. The si_signo member shall always be equal to SIGCHLD.

RETURN VALUE
If WNOHANG was specified and there are no children to wait for, 0 shall be returned. If waitid() returns due to the change of state of one of its children, 0 shall be returned. Otherwise, −1 shall be returned and errno set to indicate the error.

ERRORS
The waitid() function shall fail if:

[ECHILD] The calling process has no existing unwaited-for child processes.
[EINTR] The waitid() function was interrupted by a signal.
[EINVAL] An invalid value was specified for options, or idtype and id specify an invalid
set of processes.
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
exec, exit(), wait(), the Base Definitions volume of IEEE Std 1003.1-2001, <sys/wait.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 5
Moved from X/OPEN UNIX extension to BASE.

Issue 6
The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
waitpid — wait for a child process to stop or terminate

SYNOPSIS
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);

DESCRIPTION
Refer to wait().
NAME
wcrtomb — convert a wide-character code to a character (restartable)

SYNOPSIS
#include <stdio.h>

size_t wcrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);

DESCRIPTION
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.

If s is a null pointer, the wcrtomb() function shall be equivalent to the call:

wcrtomb(buf, L’\0’, ps)

where buf is an internal buffer.

If s is not a null pointer, the wcrtomb() function shall determine the number of bytes needed to represent the character that corresponds to the wide character given by wc (including any shift sequences), and store the resulting bytes in the array whose first element is pointed to by s. At most {MB_CUR_MAX} bytes are stored. If wc is a null wide character, a null byte shall be stored, preceded by any shift sequence needed to restore the initial shift state. The resulting state described shall be the initial conversion state.

If ps is a null pointer, the wcrtomb() 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 wcrtomb().

If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS functions, the application shall ensure that the wcrtomb() function is called with a non-NULL ps argument.

The behavior of this function shall be affected by the LC_CTYPE category of the current locale.

RETURN VALUE
The wcrtomb() function shall return the number of bytes stored in the array object (including any shift sequences). When wc is not a valid wide character, an encoding error shall occur. In this case, the function shall store the value of the macro [EILSEQ] in errno and shall return (size_t)−1; the conversion state shall be undefined.

ERRORS
The wcrtomb() function may fail if:

CX [EINVAL] ps points to an object that contains an invalid conversion state.

CX [EILSEQ] Invalid wide-character code is detected.
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
mbsinit(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY

Issue 6
In the DESCRIPTION, a note on using this function in a threaded application is added.
Extensions beyond the ISO C standard are marked.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The wcrtomb() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
wcscat — concatenate two wide-character strings

SYNOPSIS
#include <wchar.h>

wchar_t *wcscat(wchar_t *restrict ws1, const wchar_t *restrict ws2);

DESCRIPTION
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 wcscat() function shall append a copy of the wide-character string pointed to by ws2 (including the terminating null wide-character code) to the end of the wide-character string pointed to by ws1. The initial wide-character code of ws2 shall overwrite the null wide-character code at the end of ws1. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE
The wcscat() function shall return ws1; no return value is reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcsncat(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.

Issue 6
The Open Group Corrigendum U040/2 is applied. In the RETURN VALUE section, s1 is changed to ws1.
The wcscat() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
wcschr() — wide-character string scanning operation

#include <wchar.h>

wchar_t *wcschr(const wchar_t *ws, wchar_t wc);

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.

Upon completion, wcschr() shall return a pointer to the wide-character code, or a null pointer if the wide-character code is not found.

No errors are defined.

None.

None.

None.

None.

None.

wcsrchr(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

First released in Issue 4. Derived from the MSE working draft.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
wcscmp — compare two wide-character strings

SYNOPSIS
#include <wchar.h>

int wcscmp(const wchar_t *ws1, const wchar_t *ws2);

DESCRIPTION
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

The wcscmp() function shall compare the wide-character string pointed to by ws1 to the wide-
character string pointed to by ws2.

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.

RETURN VALUE
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.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcscmp(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.
NAME
wcscoll — wide-character string comparison using collating information

SYNOPSIS
#include <wchar.h>

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);

DESCRIPTION
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 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.

The wcscoll() function shall not change the setting of errno if successful.

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.

RETURN VALUE
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.

ERRORS
The wcscoll() function may fail if:

[EINVAL] The ws1 or ws2 arguments contain wide-character codes outside the domain of the collating sequence.

EXAMPLES
None.

APPLICATION USAGE
The wcsxfrm() and wcscmp() functions should be used for sorting large lists.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcscmp(), wcsxfrm(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.

Issue 5
Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

The DESCRIPTION is updated to indicate that errno is not changed if the function is successful.
NAME
wcscpy — copy a wide-character string

SYNOPSIS
#include <wchar.h>

wchar_t *wcscpy(wchar_t *restrict ws1, const wchar_t *restrict ws2);

DESCRIPTION
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 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.

RETURN VALUE
The wcscpy() function shall return ws1; no return value is reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcscpy(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.

Issue 6
The wcscpy() prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.
NAME
wcscspn — get the length of a complementary wide substring

SYNOPSIS
#include <wchar.h>

size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);

DESCRIPTION
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

The wcscspn() 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 not from the wide-character string pointed to by ws2.

RETURN VALUE
The wcscspn() function shall return the length of the initial substring of ws1; no return value is
reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcsspnt(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.

Issue 5
The RETURN VALUE section is updated to indicate that wcscspn() returns the length of ws1,
rather than ws1 itself.
NAME
wcsftime — convert date and time to a wide-character string

SYNOPSIS
#include <wchar.h>

size_t wcsftime(wchar_t *restrict wcs, size_t maxsize,
    const wchar_t *restrict format, const struct tm *restrict timeptr);

DESCRIPTION
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
The wcsftime() function shall be equivalent to the strftime() function, except that:

• The argument wcs points to the initial element of an array of wide characters into which the
generated output is to be placed.
• The argument maxsize indicates the maximum number of wide characters to be placed in the
output array.
• The argument format is a wide-character string and the conversion specifications are replaced
by corresponding sequences of wide characters.
• The return value indicates the number of wide characters placed in the output array.

If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE
If the total number of resulting wide-character codes including the terminating null wide-
character code is no more than maxsize, wcsftime() shall return the number of wide-character
codes placed into the array pointed to by wcs, not including the terminating null wide-character
code. Otherwise, zero is returned and the contents of the array are unspecified.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
strftime(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4.

Issue 5
Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of the format
argument is changed from const char * to const wchar_t *.
The `wcsftime()` prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.
NAME
wcslen — get wide-character string length

SYNOPSIS
#include <wchar.h>

size_t wcslen(const wchar_t *ws);

DESCRIPTION
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 wcslen() function shall compute the number of wide-character codes in the wide-character string to which ws points, not including the terminating null wide-character code.

RETURN VALUE
The wcslen() function shall return the length of ws; no return value is reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.
NAME
wcscat — concatenate a wide-character string with part of another

SYNOPSIS

#include <wchar.h>

wchar_t *wcscat (wchar_t *restrict ws1, const wchar_t *restrict ws2, size_t n);

DESCRIPTION

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 wcscat() function shall append not more than \( n \) wide-character codes (a null wide-character code and wide-character codes that follow it are not appended) from the array pointed to by \( ws2 \) to the end of the wide-character string pointed to by \( ws1 \). The initial wide-character code of \( ws2 \) shall overwrite the null wide-character code at the end of \( ws1 \). A terminating null wide-character code shall always be appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The wcscat() function shall return \( ws1 \); no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscat(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The wcscat() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
wcsncmp — compare part of two wide-character strings

SYNOPSIS
#include <wchar.h>

int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);

DESCRIPTION
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
The wcsncmp() function shall compare not more than n wide-character codes (wide-character
codes that follow a null wide-character code are not compared) from the array pointed to by ws1
to the array pointed to by ws2.
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.

RETURN VALUE
Upon successful completion, wcsncmp() shall return an integer greater than, equal to, or less
than 0, if the possibly null-terminated array pointed to by ws1 is greater than, equal to, or less
than the possibly null-terminated array pointed to by ws2, respectively.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcsncmp(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.
wcsncpy() — copy part of a wide-character string

#include <wchar.h>

wchar_t *wcsncpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,
                 size_t n);

DESCRIPTION
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

The wcsncpy() function shall copy not more than n wide-character codes (wide-character codes
that follow a null wide-character code are not copied) from the array pointed to by ws2 to the
array pointed to by ws1. If copying takes place between objects that overlap, the behavior is
undefined.

If the array pointed to by ws2 is a wide-character string that is shorter than n wide-character
codes, null wide-character codes shall be appended to the copy in the array pointed to by ws1,
until n wide-character codes in all are written.

RETURN VALUE
The wcsncpy() function shall return ws1; no return value is reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
If there is no null wide-character code in the first n wide-character codes of the array pointed to
by ws2, the result is not null-terminated.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcsncpy(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.

Issue 6
The wcsncpy() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
wcspbrk — scan a wide-character string for a wide-character code

SYNOPSIS
#include <wchar.h>
wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);

DESCRIPTION
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 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.

RETURN VALUE
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.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcschr(), wcsrchr(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.
**wcsrchr()**

**NAME**
wcsrchr — wide-character string scanning operation

**SYNOPSIS**

```c
#include <wchar.h>

wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);
```

**DESCRIPTION**

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 `wcsrchr()` function shall locate the last 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 shall be considered to be part of the wide-character string.

**RETURN VALUE**

Upon successful completion, `wcsrchr()` shall return a pointer to the wide-character code or a null pointer if `wc` does not occur in the wide-character string.

**ERRORS**

No errors are defined.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`wcschr()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`

**CHANGE HISTORY**

First released in Issue 4. Derived from the MSE working draft.

**Issue 6**

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
wcsrtombs — convert a wide-character string to a character string (restartable)

SYNOPSIS
#include <wchar.h>

size_t wcsrtombs(char *restrict dst, const wchar_t **restrict src, size_t len, mbstate_t *restrict ps);

DESCRIPTION

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 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)

Each conversion shall take place as if by a call to the wcrtomb() function.

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.

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().

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-NULL ps argument.

The behavior of this function shall be affected by the LC_CTYPE category of the current locale.

RETURN VALUE

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).

ERRORS

The wcsrtombs() function may fail if:

- [EINVAL] ps points to an object that contains an invalid conversion state.
- [EILSEQ] A wide-character code does not correspond to a valid character.
**wcsrtombs()**

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`mbsinit()`, `wcrtomb()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`

**CHANGE HISTORY**

Issue 6
In the DESCRIPTION, a note on using this function in a threaded application is added.
Extensions beyond the ISO C standard are marked.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The `wcsrtombs()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
wcsspn — get the length of a wide substring

SYNOPSIS
#include <wchar.h>
size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);

DESCRIPTION
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
The wcsspn() 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.

RETURN VALUE
The wcsspn() function shall return the length of the initial substring of ws1; no return value is
reserved to indicate an error.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcsspnm(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.

Issue 5
The RETURN VALUE section is updated to indicate that wcsspn() returns the length of ws1
rather that ws1 itself.
NAME
wcsstr — find a wide-character substring

SYNOPSIS
#include <wchar.h>

wchar_t *wcsstr(const wchar_t *restrict ws1,
    const wchar_t *restrict ws2);

DESCRIPTION
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 wcsstr() function shall locate the first occurrence in the wide-character string pointed to by ws1 of the sequence of wide characters (excluding the terminating null wide character) in the wide-character string pointed to by ws2.

RETURN VALUE
Upon successful completion, wcsstr() shall return a pointer to the located wide-character string, or a null pointer if the wide-character string is not found.

If ws2 points to a wide-character string with zero length, the function shall return ws1.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcschr(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY

Issue 6
The wcsstr() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
wcstod, wcstof, wcstold — convert a wide-character string to a double-precision number

SYNOPSIS
#include <wchar.h>

double wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
long double wcstold(const wchar_t * restrict nptr,
                    wchar_t ** restrict endptr);

DESCRIPTION

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

These functions shall convert the initial portion of the wide-character string pointed to by nptr to
double, float, and long double representation, respectively. First, they shall decompose the
input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by
   iswspace())
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final wide-character string of one or more unrecognized wide-character codes, including
   the terminating null wide-character code of the input wide-character string

Then they shall attempt to convert the subject sequence to a floating-point number, and return
the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the
following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an
  optional exponent part
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix
  character, then an optional binary exponent part
- One of INF or INFINITY, or any other wide string equivalent except for case
- One of NAN or NAN(n-wchar-sequenceopt), or any other wide string ignoring case in the NAN
  part, where:

  n-wchar-sequence:
  
  digit
  
  nondigit
  
  n-wchar-sequence digit
  
  n-wchar-sequence nondigit

The subject sequence is defined as the longest initial subsequence of the input wide string,
starting with the first non-white-space wide character, that is of the expected form. The subject
sequence contains no wide characters if the input wide string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of wide
characters starting with the first digit or the radix character (whichever occurs first) shall be
interpreted as a floating constant according to the rules 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 shall be 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 wide-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 wide-character
sequence NAN or NAN(n-wchar-sequenceopt) 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-wchar sequences is implementation-defined. A pointer to the final wide
string shall be 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
conversion shall be rounded in an implementation-defined manner.

The radix character shall be as 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 nptr shall be stored in the object pointed to by endptr, provided that
endptr is not a null pointer.

The wcstod() 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 wcstod(), wcstof(), or wcstold(), then check errno.

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, plus or minus 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 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].

ERRORS

The wcstod() function shall fail if:

[ERANGE] The value to be returned would cause overflow or underflow.

The wcstod() function may fail if:

[CX] [EINVAL] No conversion could be performed.
EXAMPLES
None.

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 <= D <= 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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
iswspace(), localeconv(), scanf(), setlocale(), wcstol(), the Base Definitions volume of
IEEE Std 1003.1-2001, Chapter 7, Locale, <float.h>, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.

Issue 5
The DESCRIPTION is updated to indicate that errno is not changed if the function is successful.

Issue 6
Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the
Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is
  added if no conversion could be performed.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The wcstod() prototype is updated.
- The wcstof() and wcstold() functions are added.
- If the correct value for wcstod() would cause underflow, the return value changed from 0 (as
  specified in Issue 5) to the smallest normalized positive number.
- The DESCRIPTION, RETURN VALUE, and APPLICATION USAGE sections are extensively
  updated.

NAME
wcstoiimax, wcstoumax — convert a wide-character string to an integer type

SYNOPSIS
#include <stddef.h>
#include <inttypes.h>

intmax_t wcstoiimax(const wchar_t *restrict nptr,
                     wchar_t **restrict endptr, int base);

uintmax_t wcstoumax(const wchar_t *restrict nptr,
                     wchar_t **restrict endptr, int base);

DESCRIPTION
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

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.

RETURN VALUE
These functions shall return the converted value, if any.

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}.

ERRORS
These functions shall fail if:

[EINVAL] The value of base is not supported.

[ERANGE] The value to be returned is not representable.

These functions may fail if:

[EINVAL] No conversion could be performed.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcstol(), wcstoul(), the Base Definitions volume of IEEE Std 1003.1-2001, <inttypes.h>,
<stddef.h>

CHANGE HISTORY
wcstok()
EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
The Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4.

Issue 5
Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, a third argument is added to the definition of wcstok() in the SYNOPSIS.

Issue 6
The wcstok() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME
wcstol, wcstoll — convert a wide-character string to a long integer

SYNOPSIS
#include <wchar.h>

long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);

long long wcstoll(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);

DESCRIPTION
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.

These functions shall convert the initial portion of the wide-character string pointed to by nptr to long, long long, and unsigned long long representation, respectively. First, they shall decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by iswspace())
2. A subject sequence interpreted as an integer represented in some radix determined by the value of base
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string

Then they shall attempt to convert the subject sequence to an integer, and return the result.

If 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 begins with 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.

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, but not including an integer suffix. 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 shall be permitted. If the value of base is 16, the wide-character code representations of 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first non-white-space wide-character code that is of the expected form. The subject sequence contains no wide-character codes if the input wide-character string is empty or consists entirely of white-space wide-character code, or if the first non-white-space wide-character code is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and base is 0, the sequence of wide-character codes 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 wide-character string shall be stored in the object pointed to by endptr, provided that endptr is
not a null pointer.

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 nptr shall be stored in the object pointed to by endptr, provided that endptr is not a null pointer.

These functions shall not change the setting of errno if successful.

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 should set errno to 0, then call wcstol() or wcstoll(), then check errno.

RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no conversion could be performed, 0 shall be returned and errno may be set to indicate the error. If the correct value is outside the range of representable values, [LONG_MIN], [LONG_MAX], [LLONG_MIN], or [LLONG_MAX] shall be returned (according to the sign of the value), and errno set to [ERANGE].

ERRORS

These functions shall fail if:

- [EINVAL] The value of base is not supported.
- [ERANGE] The value to be returned is not representable.

These functions may fail if:

- [EINVAL] No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalpha(), scanf(), wcstod(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The DESCRIPTION is updated to indicate that errno is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.
The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The `wcstol()` prototype is updated.
- The `wcstoll()` function is added.
NAME
wcstold — convert a wide-character string to a double-precision number

SYNOPSIS
#include <wchar.h>

long double wcstold(const wchar_t * restrict nptr,
                      wchar_t ** restrict endptr);

DESCRIPTION
Refer to wcstod().
NAME
wcstoll — convert a wide-character string to a long integer

SYNOPSIS
#include <wchar.h>

long long wcstoll(const wchar_t *restrict nptr,
                   wchar_t **restrict endptr, int base);

DESCRIPTION
Refer to wcstol().
NAME
wcstombs — convert a wide-character string to a character string

SYNOPSIS
#include <stdlib.h>

size_t wcstombs(char *restrict s, const wchar_t *restrict pwcs,
        size_t n);

DESCRIPTION
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

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.

The behavior of this function shall be affected by the LC_CTYPE category of the current locale.

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.

The wcstombs() function need not be reentrant. A function that is not required to be reentrant is
not required to be thread-safe.

RETURN VALUE
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.

ERRORS
The wcstombs() function may fail if:

EILSEQ A wide-character code does not correspond to a valid character.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
mblen(), mbtowc(), mbstowcs(), wctomb(), the Base Definitions volume of IEEE Std 1003.1-2001,
<stdlib.h>
First released in Issue 4. Derived from the ISO C standard.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION states the effect of when \texttt{s} is a null pointer.
- The [EILSEQ] error condition is added.

The \texttt{wcstombs()} prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.
NAME
wcstoul, wcstoull — convert a wide-character string to an unsigned long

SYNOPSIS
#include <wchar.h>
unsigned long wcstoul(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
unsigned long long wcstoull(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);

DESCRIPTION
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 wcstoul() and wcstoull() functions shall convert the initial portion of the wide-character string pointed to by nptr to unsigned long and unsigned long long representation, respectively. First, they shall decompose the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by iswspace())
2. A subject sequence interpreted as an integer represented in some radix determined by the value of base
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string

Then they shall attempt to convert the subject sequence to an unsigned integer, and return the result.

If 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.

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, but not including an integer suffix. 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 shall be permitted. If the value of base is 16, the wide-character codes 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first wide-character code that is not white space and is of the expected form. The subject sequence contains no wide-character codes if the input wide-character string is empty or consists entirely of white-space wide-character codes, or if the first wide-character code that is not white space is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and base is 0, the sequence of wide-character codes 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 wide-character string shall be stored in the object pointed to by endptr, provided that endptr is
not a null pointer.

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 `nptr` shall be stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

The `wcstoul()` function shall not change the setting of `errno` if successful.

Since 0, [ULONG_MAX], and [ULLONG_MAX] are returned on error and 0 is also a valid return on success, an application wishing to check for error situations should set `errno` to 0, then call `wcstoul()` or `wcstoull()`, then check `errno`.

Upon successful completion, the `wcstoul()` and `wcstoull()` functions shall return the converted value, if any. If no conversion could be performed, 0 shall be returned and `errno` may be set to indicate the error. If the correct value is outside the range of representable values, [ULONG_MAX] or [ULLONG_MAX] respectively shall be returned and `errno` set to [ERANGE].

These functions shall fail if:

- [EINVAL] The value of `base` is not supported.

- [ERANGE] The value to be returned is not representable.

These functions may fail if:

- [EINVAL] No conversion could be performed.

Examples

None.

Application Usage

None.

Rationale

None.

Future Directions

None.

See Also

`iswalpha()`, `scanf()`, `wcstod()`, `wcstol()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`

Change History

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] error condition is added for when the value of `base` is not supported.
In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:
- The \texttt{wcstoul()} prototype is updated.
- The \texttt{wcstoull()} function is added.
NAME
wcstoumax — convert a wide-character string to an integer type

SYNOPSIS
#include <stddef.h>
#include <inttypes.h>

uintmax_t wcstoumax(const wchar_t *restrict nptr,
                     wchar_t **restrict endptr, int base);

DESCRIPTION
Refer to wcstoi().
NAME
wcswcs — find a wide substring (LEGACY)

SYNOPSIS
#include <wchar.h>
wchar_t *wcswcs(const wchar_t *ws1, const wchar_t *ws2);

DESCRIPTION
The wcswcs() 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.

RETURN VALUE
Upon successful completion, wcswcs() shall return a pointer to the located wide-character string
or a null pointer if the wide-character string is not found.

If ws2 points to a wide-character string with zero length, the function shall return ws1.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
This function was not included in the final ISO/IEC 9899:1990/Amendment 1:1995 (E).
Application developers are strongly encouraged to use the wcsstr() function instead.

RATIONALE
None.

FUTURE DIRECTIONS
This function may be withdrawn in a future version.

SEE ALSO
wcschr(), wcsstr(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.

Issue 5
Marked EX.

Issue 6
This function is marked LEGACY.
NAME
wcswidth — number of column positions of a wide-character string

SYNOPSIS
XSI
#include <wchar.h>

int wcswidth(const wchar_t *pwcs, size_t n);

DESCRIPTION
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 \( pwcs \).

RETURN VALUE
The wcswidth() function either shall return 0 (if \( pwcs \) points to a null wide-character code), or return the number of column positions to be occupied by the wide-character string pointed to by \( pwcs \), or return \(-1\) (if any of the first \( n \) wide-character codes in the wide-character string pointed to by \( pwcs \) is not a printable wide-character code).

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the return value for a non-printable wide character is not specified.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcwidth(), the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.103, Column Position, <wchar.h>

CHANGE HISTORY
First released in Issue 4. Derived from the MSE working draft.

Issue 6
The Open Group Corrigendum U021/11 is applied. The function is marked as an extension.
NAME
wcsxfrm — wide-character string transformation

SYNOPSIS
#include <wchar.h>

size_t wcsxfrm(wchar_t *restrict ws1, const wchar_t *restrict ws2,
size_t n);

DESCRIPTION
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

The wcsxfrm() function shall transform the wide-character string pointed to by ws2 and place the
resulting wide-character string into the array pointed to by ws1. The transformation shall be
such that if wcscmp() is applied to two transformed wide strings, it shall return a value greater
than, equal to, or less than 0, corresponding to the result of wcscoll() applied to the same two
original wide-character strings. No more than n wide-character codes shall be placed into the
resulting array pointed to by ws1, including the terminating null wide-character code. If n is 0,
ws1 is permitted to be a null pointer. If copying takes place between objects that overlap, the
behavior is undefined.

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 wcsxfrm(), then check errno.

RETURN VALUE
The wcsxfrm() function shall return the length of the transformed wide-character string (not
including the terminating null wide-character code). If the value returned is n or more, the
contents of the array pointed to by ws1 are unspecified.

On error, the wcsxfrm() function may set errno, but no return value is reserved to indicate an
error.

ERRORS
The wcsxfrm() function may fail if:

[EINVAL] The wide-character string pointed to by ws2 contains wide-character codes
outside the domain of the collating sequence.

EXAMPLES
None.

APPLICATION USAGE
The transformation function is such that two transformed wide-character strings can be ordered
by wcscmp() as appropriate to collating sequence information in the program’s locale (category
LC_COLLATE).

The fact that when n is 0 ws1 is permitted to be a null pointer is useful to determine the size of
the ws1 array prior to making the transformation.

RATIONALE
None.
**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`wcsncmp()`, `wcscoll()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`

**CHANGE HISTORY**

First released in Issue 4. Derived from the MSE working draft.

**Issue 5**

Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

The DESCRIPTION is updated to indicate that `errno` is not changed if the function is successful.

**Issue 6**

In previous versions, this function was required to return −1 on error.

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The `wcsxfrm()` prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
**NAME**
wctob — wide-character to single-byte conversion

**SYNOPSIS**
```c
#include <stdio.h>
#include <wchar.h>
int wctob(wint_t c);
```

**DESCRIPTION**
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 `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.

The behavior of this function shall be affected by the `LC_CTYPE` category of the current locale.

**RETURN VALUE**
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**.

**ERRORS**
No errors are defined.

**EXAMPLES**
None.

**APPLICATION USAGE**
None.

**RATIONALE**
None.

**FUTURE DIRECTIONS**
None.

**SEE ALSO**
`btowc()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`

**CHANGE HISTORY**
NAME
wctomb — convert a wide-character code to a character

SYNOPSIS
#include <stdlib.h>

int wctomb(char *s, wchar_t wchar);

DESCRIPTION
The wctomb() function shall determine the number of bytes needed to represent the character corresponding to the wide-character code whose value is wchar (including any change in the shift state). It shall store the character representation (possibly multiple bytes and any special bytes to change shift state) in the array object pointed to by s (if s is not a null pointer). At most \{MB_CUR_MAX\} bytes shall be stored. If wchar is 0, a null byte shall be stored, preceded by any shift sequence needed to restore the initial shift state, and wctomb() shall be left in the initial shift state.

The behavior of this function is affected by the LC_CTYPE category of the current locale. For a state-dependent encoding, this function shall be 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. Changing the LC_CTYPE category causes the shift state of this function to be unspecified.

The wctomb() function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-2001 calls wctomb().

RETURN VALUE
If s is a null pointer, wctomb() 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, wctomb() shall return -1 if the value of wchar does not correspond to a valid character, or return the number of bytes that constitute the character corresponding to the value of wchar.

In no case shall the value returned be greater than the value of the \{MB_CUR_MAX\} macro.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.
SEE ALSO

`mblen()`, `mbtowc()`, `mbstowcs()`, `wcstombs()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<stdlib.h>`

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Extensions beyond the ISO C standard are marked.

In the DESCRIPTION, a note about reentrancy and thread-safety is added.
NAME
wctrans — define character mapping

SYNOPSIS
#include <wctype.h>
wctrans_t wctrans(const char *charclass);

DESCRIPTION
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 wctrans() function is defined for valid character mapping names identified in the current locale. The charclass is a string identifying a generic character mapping name for which codeset-specific information is required. The following character mapping names are defined in all locales: tolower and toupper.

The function shall return a value of type wctrans_t, which can be used as the second argument to subsequent calls of towctrans(). The wctrans() function shall determine values of wctrans_t according to the rules of the coded character set defined by character mapping information in the program's locale (category LC_CTYPE). The values returned by wctrans() shall be valid until a call to setlocale() that modifies the category LC_CTYPE.

RETURN VALUE
The wctrans() function shall return 0 and may set errno to indicate the error if the given character mapping name is not valid for the current locale (category LC_CTYPE); otherwise, it shall return a non-zero object of type wctrans_t that can be used in calls to towctrans().

ERRORS
The wctrans() function may fail if:

[EINVAL] The character mapping name pointed to by charclass is not valid in the current locale.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
towctrans(), the Base Definitions volume of IEEE Std 1003.1-2001, <wctype.h>

CHANGE HISTORY
NAME
wctype — define character class

SYNOPSIS
#include <wctype.h>
wctype_t wctype(const char *property);

DESCRIPTION
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
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:

   alnum  digit  punct
   alpha  graph  space
   blank  lower  upper
   cntrl  print  xdigit

Additional character class names defined in the locale definition file (category LC_CTYPE) can
also be specified.
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.

RETURN VALUE
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().

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
iswctype(), the Base Definitions volume of IEEE Std 1003.1-2001, <wctype.h>

CHANGE HISTORY
First released in Issue 4.
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>`.
NAME
wcwidth — number of column positions of a wide-character code

SYNOPSIS
XSI
#include <wchar.h>

int wcwidth(wchar_t wc);

DESCRIPTION
The wcwidth() function shall determine the number of column positions required for the wide character wc. The application shall ensure that the value of wc is a character representable as a wchar_t, and is a wide-character code corresponding to a valid character in the current locale.

RETURN VALUE
The wcwidth() function shall either return 0 (if wc is a null wide-character code), or return the number of column positions to be occupied by the wide-character code wc, or return −1 (if wc does not correspond to a printable wide-character code).

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the return value for a non-printable wide character is not specified.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wcwidth(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY
First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working draft.

Issue 6
The Open Group Corrigendum U021/12 is applied. This function is marked as an extension.
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
**NAME**

wmemchr — find a wide character in memory

**SYNOPSIS**

```c
#include <wchar.h>

wchar_t *wmemchr(const wchar_t *ws, wchar_t wc, size_t n);
```

**DESCRIPTION**

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 `wmemchr()` function shall locate the first occurrence of `wc` in the initial `n` wide characters of the object pointed to by `ws`. 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.

If `n` is zero, the application shall ensure that `ws` is a valid pointer and the function behaves as if no valid occurrence of `wc` is found.

**RETURN VALUE**

The `wmemchr()` function shall return a pointer to the located wide character, or a null pointer if the wide character does not occur in the object.

**ERRORS**

No errors are defined.

**EXAMPLES**

None.

**APPLICATION USAGE**

None.

**RATIONALE**

None.

**FUTURE DIRECTIONS**

None.

**SEE ALSO**

`wmemcmp()`, `wmemcmp()`, `wmemmove()`, `wmemset()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`

**CHANGE HISTORY**


**Issue 6**

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
wmemcmp — compare wide characters in memory

SYNOPSIS
#include <wchar.h>

int wmemcmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);

DESCRIPTION
 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.

The wmemcmp() function shall compare the first n wide characters of the object pointed to by ws1 to the first n wide characters of the object pointed to by ws2. 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.

If n is zero, the application shall ensure that ws1 and ws2 are valid pointers, and the function shall behave as if the two objects compare equal.

RETURN VALUE
The wmemcmp() function shall return an integer greater than, equal to, or less than zero, respectively, as the object pointed to by ws1 is greater than, equal to, or less than the object pointed to by ws2.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wmemchr(), wmemcpy(), wmemmove(), wmemset(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME

wmemcpy — copy wide characters in memory

SYNOPSIS

#include <wchar.h>

wchar_t *wmemcpy(wchar_t *restrict ws1, const wchar_t *restrict ws2, size_t n);

DESCRIPTION

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 wmemcpy() function shall copy n wide characters from the object pointed to by ws2 to the object pointed to by ws1. 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.

If n is zero, the application shall ensure that ws1 and ws2 are valid pointers, and the function shall copy zero wide characters.

RETURN VALUE

The wmemcpy() function shall return the value of ws1.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wmemchr(), wmemcmp(), wmemmove(), wmemset(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY


Issue 6

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The wmemcpy() prototype is updated for alignment with the ISO/IEC 9899:1999 standard.
NAME

wmemmove — copy wide characters in memory with overlapping areas

SYNOPSIS

#include <wchar.h>

wchar_t *wmemmove(wchar_t *ws1, const wchar_t *ws2, size_t n);

DESCRIPTION

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 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.

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.

If n is zero, the application shall ensure that ws1 and ws2 are valid pointers, and the function shall copy zero wide characters.

RETURN VALUE

The wmemmove() function shall return the value of ws1.

ERRORS

No errors are defined

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wmemchr(), wmemcmp(), wmemcpy(), wmemset(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY


Issue 6

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
NAME
wnemset — set wide characters in memory

SYNOPSIS
#include <wchar.h>

wchar_t *wmemset(wchar_t *ws, wchar_t wc, size_t n);

DESCRIPTION
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 wmemset() function shall copy the value of wc into each of the first n wide characters of the object pointed to by ws. 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.

If n is zero, the application shall ensure that ws is a valid pointer, and the function shall copy zero wide characters.

RETURN VALUE
The wmemset() functions shall return the value of ws.

ERRORS
No errors are defined.

EXAMPLES
None.

APPLICATION USAGE
None.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
wmemchr(), wmemcmp(), wmemcpy(), wmemmove(), the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>

CHANGE HISTORY

Issue 6
The DESCRIPTION is updated to avoid use of the term "must" for application requirements.
NAME
wordexp, wordfree — perform word expansions

SYNOPSIS
#include <wordexp.h>

int wordexp(const char *restrict words, wordexp_t *restrict pwordexp, int flags);
void wordfree(wordexp_t *pwordexp);

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:

<table>
<thead>
<tr>
<th>Member Type</th>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t</td>
<td>we_wordc</td>
<td>Count of words matched by words.</td>
</tr>
<tr>
<td>char **</td>
<td>we_wordv</td>
<td>Pointer to list of expanded words.</td>
</tr>
<tr>
<td>size_t</td>
<td>we_offs</td>
<td>Slots to reserve at the beginning of pwordexp-&gt;we_wordv.</td>
</tr>
</tbody>
</table>

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_words. The wordfree() function frees any memory associated with pwordexp from a previous call to wordexp().
The *flags* argument is used to control the behavior of *wordexp()*.

The value of *flags* is the bitwise-inclusive OR of zero or more of the following constants, which are defined in `<wordexp.h>`:

- **WRDE_APPEND** Append words generated to the ones from a previous call to *wordexp()*.
- **WRDE_DOOFFS** Make use of *pwordexp-*`->we_offs`. If this flag is set, *pwordexp-*`->we_offs` is used to specify how many null pointers to add to the beginning of *pwordexp-*`->we_wordv`. In other words, *pwordexp-*`->we_wordv` shall point to *pwordexp-*`->we_offs` null pointers, followed by *pwordexp-*`->we_wordc` word pointers, followed by a null pointer.
- **WRDE_NOCMD** If the implementation supports the utilities defined in the Shell and Utilities volume of IEEE Std 1003.1-2001, fail if command substitution, as specified in the Shell and Utilities volume of IEEE Std 1003.1-2001, Section 2.6.3, Command Substitution, is requested.
- **WRDE_REUSE** The *pwordexp* argument was passed to a previous successful call to *wordexp()* and has not been passed to *wordfree()*.
  
  The result shall be the same as if the application had called *wordfree()* and then called *wordexp()* without WRDE_REUSE.
- **WRDE_SHOWERERR** Do not redirect *stderr* to `/dev/null`.
- **WRDE_UNDEF** Report error on an attempt to expand an undefined shell variable.

The WRDE_APPEND flag can be used to append a new set of words to those generated by a previous call to *wordexp()*.

The following rules apply to applications when two or more calls to *wordexp()* are made with the same value of *pwordexp* and without intervening calls to *wordfree()*:

1. The first such call shall not set WRDE_APPEND. All subsequent calls shall set it.
2. All of the calls shall set WRDE_DOOFFS, or all shall not set it.
3. After the second and each subsequent call, *pwordexp-*`->we_wordv` shall point to a list containing the following:
   a. Zero or more null pointers, as specified by WRDE_DOOFFS and *pwordexp-*`->we_offs`
   b. Pointers to the words that were in the *pwordexp-*`->we_wordv` list before the call, in the same order as before
   c. Pointers to the new words generated by the latest call, in the specified order
4. The count returned in *pwordexp-*`->we_wordc` shall be the total number of words from all of the calls.
5. The application can change any of the fields after a call to *wordexp()* but if it does it shall reset them to the original value before a subsequent call, using the same *pwordexp* value, to *wordfree()* or *wordexp()* with the WRDE_APPEND or WRDE_REUSE flag.

If the implementation supports the utilities defined in the Shell and Utilities volume of IEEE Std 1003.1-2001, and *words* contains an unquoted character—`<newline>`, `'`, `'`, `'`, `'`, `'`—in an inappropriate context, *wordexp()* shall fail, and the number of expanded words shall be 0.

Unless WRDE_SHOWERERR is set in *flags*, *wordexp()* shall redirect *stderr* to `/dev/null` for any utilities executed as a result of command substitution while expanding *words*. If WRDE_SHOWERERR is set, *wordexp()* may write messages to *stderr* if syntax errors are detected while expanding *words*. 

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved. 1645
The application shall ensure that if WRDE_DOFFS is set, then `pwordexp->we_offs` has the same value for each `wordexp()` call and `wordfree()` call using a given `pwordexp`.

The following constants are defined as error return values:

- **WRDE_BADCHAR**: One of the unquoted characters—`<newline>`, `'`, `&`, `;`, `'`, `'`, `'`, `'`, `|`, `'`—appears in `words` in an inappropriate context.
- **WRDE_BADVAL**: Reference to undefined shell variable when WRDE_UNDEF is set in `flags`.
- **WRDE_CMDSUB**: Command substitution requested when WRDE_NOCMD was set in `flags`.
- **WRDE_NOSPACE**: Attempt to allocate memory failed.
- **WRDE_SYNTAX**: Shell syntax error, such as unbalanced parentheses or unterminated string.

**RETURN VALUE**

Upon successful completion, `wordexp()` shall return 0. Otherwise, a non-zero value, as described in `<wordexp.h>`, shall be returned to indicate an error. If `wordexp()` returns the value `WRDE_NOSPACE`, then `pwordexp->we_wordc` and `pwordexp->we_wordv` shall be updated to reflect any words that were successfully expanded. In other cases, they shall not be modified.

The `wordfree()` function shall not return a value.

**ERRORS**

No errors are defined.

**EXAMPLES**

None.

**APPLICATION USAGE**

The `wordexp()` function is intended to be used by an application that wants to do all of the shell’s expansions on a word or words obtained from a user. For example, if the application prompts for a filename (or list of filenames) and then uses `wordexp()` to process the input, the user could respond with anything that would be valid as input to the shell.

The WRDE_NOCMD flag is provided for applications that, for security or other reasons, want to prevent a user from executing shell commands. Disallowing unquoted shell special characters also prevents unwanted side effects, such as executing a command or writing a file.

**RATIONALE**

This function was included as an alternative to `glob()`. There had been continuing controversy over exactly what features should be included in `glob()`. It is hoped that by providing `wordexp()` (which provides all of the shell word expansions, but which may be slow to execute) and `glob()` (which is faster, but which only performs pathname expansion, without tilde or parameter expansion) this will satisfy the majority of applications.

While `wordexp()` could be implemented entirely as a library routine, it is expected that most implementations run a shell in a subprocess to do the expansion.

Two different approaches have been proposed for how the required information might be presented to the shell and the results returned. They are presented here as examples.

One proposal is to extend the `echo` utility by adding a `-q` option. This option would cause `echo` to add a backslash before each backslash and `<blank>` that occurs within an argument. The `wordexp()` function could then invoke the shell as follows:

```c
(void) strcpy(buffer, "echo -q");
(void) strcat(buffer, words);
if (((flags & WRDE_SHOWERR) == 0)
```
(void) strcat(buffer, "2>/dev/null");
f = popen(buffer, "r");

The wordexp() function would read the resulting output, remove unquoted backslashes, and break into words at unquoted <blank>s. If the WRDE_NOCMD flag was set, wordexp() would have to scan words before starting the subshell to make sure that there would be no command substitution. In any case, it would have to scan words for unquoted special characters.

Another proposal is to add the following options to sh:

-w wordlist

This option provides a wordlist expansion service to applications. The words in wordlist shall be expanded and the following written to standard output:

1. The count of the number of words after expansion, in decimal, followed by a null byte
2. The number of bytes needed to represent the expanded words (not including null separators), in decimal, followed by a null byte
3. The expanded words, each terminated by a null byte

If an error is encountered during word expansion, sh exits with a non-zero status after writing the former to report any words successfully expanded

-P Run in "protected" mode. If specified with the -w option, no command substitution shall be performed.

With these options, wordexp() could be implemented fairly simply by creating a subprocess using fork() and executing sh using the line:

`exec<shell path>, "sh", "-P", "-w", words, (char *)0);`

after directing standard error to /dev/null.

It seemed objectionable for a library routine to write messages to standard error, unless explicitly requested, so wordexp() is required to redirect standard error to /dev/null to ensure that no messages are generated, even for commands executed for command substitution. The WRDE_SHOWERR flag can be specified to request that error messages be written.

The WRDE_REUSE flag allows the implementation to avoid the expense of freeing and reallocating memory, if that is possible. A minimal implementation can call wordfree() when WRDE_REUSE is set.

FUTURE DIRECTIONS

None.

SEE ALSO

fnmatch(), glob(), the Base Definitions volume of IEEE Std 1003.1-2001, <wordexp.h>, the Shell and Utilities volume of IEEE Std 1003.1-2001, Chapter 2, Shell Command Language

CHANGE HISTORY


Issue 5

Moved from POSIX2 C-language Binding to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

The restrict keyword is added to the wordexp() prototype for alignment with the ISO/IEC 9899:1999 standard.
wprintf()

NAME
wprintf — print formatted wide-character output

SYNOPSIS
#include <stdio.h>
#include <wchar.h>

int wprintf(const wchar_t *restrict format, ...);

DESCRIPTION
Refer to fwprintf().
NAME
pwrite, write — write on a file

SYNOPSIS
#include <unistd.h>

XSI
ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
    off_t offset);

ssize_t write(int fildes, const void *buf, size_t nbyte);

DESCRIPTION
The write() function shall attempt to write nbyte bytes from the buffer pointed to by buf to the file associated with the open file descriptor, fildes.

Before any action described below is taken, and if nbyte is zero and the file is a regular file, the write() function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the write() function shall return zero and have no other results.

If nbyte is zero and the file is not a regular file, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data shall proceed from the position in the file indicated by the file offset associated with fildes. Before successful return from write(), the file offset shall be incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file shall be set to this file offset.

On a file not capable of seeking, writing shall always take place starting at the current position. The value of a file offset associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file offset shall be set to the end of the file prior to each write and no intervening file modification operation shall occur between changing the file offset and the write operation.

XSI
If a write() requests that more bytes be written than there is room for (for example, the process' file size limit or the physical end of a medium), only as many bytes as there is room for shall be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes would give a failure return (except as noted below).

XSI
If the request would cause the file size to exceed the soft file size limit for the process and there is no room for any bytes to be written, the request shall fail and the implementation shall generate the SIGXFSZ signal for the thread.

If write() is interrupted by a signal before it writes any data, it shall return -1 with errno set to [EINTR].

If write() is interrupted by a signal after it successfully writes some data, it shall return the number of bytes written.

If the value of nbyte is greater than [SSIZE_MAX], the result is implementation-defined.

After a write() to a regular file has successfully returned:

- Any successful read() from each byte position in the file that was modified by that write shall return the data specified by the write() for that position until such byte positions are again modified.
- Any subsequent successful write() to the same byte position in the file shall overwrite that file data.
Write requests to a pipe or FIFO shall be handled in the same way as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request shall append to the end of the pipe.
- Write requests of \{PIPE_BUF\} bytes or less shall not be interleaved with data from other processes doing writes on the same pipe. Writes of greater than \{PIPE_BUF\} bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK flag of the file status flags is set.
- If the O_NONBLOCK flag is clear, a write request may cause the thread to block, but on normal completion it shall return \(nbyte\).
- If the O_NONBLOCK flag is set, \texttt{write()} requests shall be handled differently, in the following ways:
  - The \texttt{write()} function shall not block the thread.
  - A write request for \{PIPE_BUF\} or fewer bytes shall have the following effect: if there is sufficient space available in the pipe, \texttt{write()} shall transfer all the data and return the number of bytes requested. Otherwise, \texttt{write()} shall transfer no data and return \(-1\) with \texttt{errno} set to [EAGAIN].
  - A write request for more than \{PIPE_BUF\} bytes shall cause one of the following:
    - When at least one byte can be written, transfer what it can and return the number of bytes written. When all data previously written to the pipe is read, it shall transfer at least \{PIPE_BUF\} bytes.
    - When no data can be written, transfer no data, and return \(-1\) with \texttt{errno} set to [EAGAIN].

When attempting to write to a file descriptor (other than a pipe or FIFO) that supports non-blocking writes and cannot accept the data immediately:

- If the O_NONBLOCK flag is clear, \texttt{write()} shall block the calling thread until the data can be accepted.
- If the O_NONBLOCK flag is set, \texttt{write()} shall not block the thread. If some data can be written without blocking the thread, \texttt{write()} shall write what it can and return the number of bytes written. Otherwise, it shall return \(-1\) and set \texttt{errno} to [EAGAIN].

Upon successful completion, where \(nbyte\) is greater than 0, \texttt{write()} shall mark for update the \texttt{st_ctime} and \texttt{st_mtime} fields of the file, and if the file is a regular file, the S_ISUID and S_ISGID bits of the file mode may be cleared.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with \texttt{fildes}.

If \texttt{fildes} refers to a socket, \texttt{write()} shall be equivalent to \texttt{send()} with no flags set.

If the O_DSYNC bit has been set, write I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion.

If the O_SYNC bit has been set, write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

If \texttt{fildes} refers to a shared memory object, the result of the \texttt{write()} function is unspecified.

If \texttt{fildes} refers to a typed memory object, the result of the \texttt{write()} function is unspecified.
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.

When writing to a STREAM, data messages are created with a priority band of 0. When writing 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.

In addition, *write()* shall fail if the STREAM head has processed an asynchronous error before the call. In this case, the value of *errno* does not reflect the result of *write()*, but reflects the prior error.

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.

**RETURN VALUE**

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.

**ERRORS**

The *write()* and *pwrite()* functions shall fail if:

- [EAGAIN] The O_NONBLOCK flag is set for the file descriptor and the thread would be delayed in the *write()* operation.
- [EBADF] The *fildes* argument is not a valid file descriptor open for writing.
- [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.
- [EFBIG] The file is a regular file, *nbyte* 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 *fildes*.
- [EINVAL] The write operation was terminated due to the receipt of a signal, and no data was transferred.
- [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
write()

blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.

[ENOSPC] There was no free space remaining on the device containing the file.

[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.

[ERANGE] The transfer request size was outside the range supported by the STREAMS file associated with fildes.

The write() function shall fail if:

[EAGAIN] or [EWOULDBLOCK]
The file descriptor is for a socket, is marked O_NONBLOCK, and write would block.

[ECONNRESET] A write was attempted on a socket that is not connected.

[EPipe] A write was attempted on a socket that is shut down for writing, or is no longer connected. In the latter case, if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.

The write() and pwrite() functions may fail if:

[EINVAL] The STREAM or multiplexer referenced by fildes is linked (directly or indirectly) downstream from a multiplexer.

[EIO] A physical I/O error has occurred.

[ENOBFS] Insufficient resources were available in the system to perform the operation.

[ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

[ENXIO] A hangup occurred on the STREAM being written to.

A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, errno is set to the value included in the error message.

The write() function may fail if:

[EACCES] A write was attempted on a socket and the calling process does not have appropriate privileges.

[ENETDOWN] A write was attempted on a socket and the local network interface used to reach the destination is down.

[ENETUNREACH] A write was attempted on a socket and no route to the network is present.

The pwrite() function shall fail and the file pointer remain unchanged if:

[EINVAL] The offset argument is invalid. The value is negative.

[ESPIPE] fildes is associated with a pipe or FIFO.
EXAMPLES

Writing from a Buffer

The following example writes data from the buffer pointed to by `buf` to the file associated with the file descriptor `fd`.

```
#include <sys/types.h>
#include <string.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_written;
int fd;
...
strcpy(buf, "This is a test\n");
nbytes = strlen(buf);
bytes_written = write(fd, buf, nbytes);
...
```

APPLICATION USAGE

None.

RATIONALE

See also the RATIONALE section in `read()`.

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 process 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.

- **Complete/partial/deferred**: A write request:

  ```
  int fildes;
  size_t nbyte;
  ssize_t ret;
  char *buf;
  ret = write(fildes, buf, nbyte);
  ```

  may return:

  Complete  `ret=nbyte`

  Partial   `ret<nbyte`

  This shall never happen if `nbyte`≤ `{PIPE_BUF}`. If it does happen (with `nbyte`> `{PIPE_BUF}`), this volume of IEEE Std 1003.1-2001 does not guarantee
atomicity, even if \( \text{ret} \leq \text{PIPE_BUF} \), because atomicity is guaranteed according to the amount requested, not the amount written.

Deferred: \( \text{ret} = -1, \text{errno} = [\text{EAGAIN}] \)

This error indicates that a later request may succeed. It does not indicate that it shall succeed, even if \( nbyte \leq \text{PIPE_BUF} \), because if no process reads from the pipe or FIFO, the write never succeeds. An application could usefully count the number of times \([\text{EAGAIN}]\) is caused by a particular value of \( nbyte > \text{PIPE_BUF} \) and perhaps do later writes with a smaller value, on the assumption that the effective size of the pipe may have decreased.

Partial and deferred writes are only possible with O_NONBLOCK set.

The relations of these properties are shown in the following tables:

### Write to a Pipe or FIFO with O_NONBLOCK clear

<table>
<thead>
<tr>
<th>Immediately Writable:</th>
<th>None</th>
<th>Some ( nbyte )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( nbyte \leq \text{PIPE_BUF} )</td>
<td>Atomic blocking ( nbyte )</td>
<td>Atomic blocking ( nbyte )</td>
</tr>
<tr>
<td>( nbyte &gt; \text{PIPE_BUF} )</td>
<td>Blocking ( nbyte )</td>
<td>Blocking ( nbyte )</td>
</tr>
</tbody>
</table>

If the O_NONBLOCK flag is clear, a write request shall block if the amount writable immediately is less than that requested. If the flag is set (by \text{fcntl}()), a write request shall never block.

### Write to a Pipe or FIFO with O_NONBLOCK set

<table>
<thead>
<tr>
<th>Immediately Writable:</th>
<th>None</th>
<th>Some ( nbyte )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( nbyte \leq \text{PIPE_BUF} )</td>
<td>-1, ([\text{EAGAIN}])</td>
<td>-1, ([\text{EAGAIN}])</td>
</tr>
<tr>
<td>( nbyte &gt; \text{PIPE_BUF} )</td>
<td>-1, ([\text{EAGAIN}])</td>
<td>(&lt;nbyte \text{ or } -1)</td>
</tr>
</tbody>
</table>

There is no exception regarding partial writes when O_NONBLOCK is set. With the exception of writing to an empty pipe, this volume of IEEE Std 1003.1-2001 does not specify exactly when a partial write is performed since that would require specifying internal details of the implementation. Every application should be prepared to handle partial writes when O_NONBLOCK is set and the requested amount is greater than \text{PIPE_BUF}, just as every application should be prepared to handle partial writes on other kinds of file descriptors.

The intent of forcing writing at least one byte if any can be written is to assure that each write makes progress if there is any room in the pipe. If the pipe is empty, \text{PIPE_BUF} bytes must be written; if not, at least some progress must have been made.

Where this volume of IEEE Std 1003.1-2001 requires -1 to be returned and errno set to \([\text{EAGAIN}]\), most historical implementations return zero (with the O_NDELAY flag set, which is the historical predecessor of O_NONBLOCK, but is not itself in this volume of IEEE Std 1003.1-2001). The error indications in this volume of IEEE Std 1003.1-2001 were chosen so that an application can distinguish these cases from end-of-file. While \text{write}() cannot receive an indication of end-of-file, \text{read}() can, and the two functions have similar return values. Also, some existing systems (for example, Eighth Edition) permit a write of zero bytes to mean that the reader should get an end-of-file indication; for those systems, a return value of zero from \text{write}() indicates a successful write of an end-of-file indication.
Implementations are allowed, but not required, to perform error checking for \texttt{write()} requests of zero bytes.

The concept of a \{PIPE\_MAX\} limit (indicating the maximum number of bytes that can be written to a pipe in a single operation) was considered, but rejected, because this concept would unnecessarily limit application writing.

See also the discussion of O\_NONBLOCK in \texttt{read()}. Writes can be serialized with respect to other reads and writes. If a \texttt{read()} of file data can be proven (by any means) to occur after a \texttt{write()} of the data, it must reflect that \texttt{write()}, even if the calls are made by different processes. A similar requirement applies to multiple write operations to the same file position. This is needed to guarantee the propagation of data from \texttt{write()} calls to subsequent \texttt{read()} calls. This requirement is particularly significant for networked file systems, where some caching schemes violate these semantics.

This volume of IEEE Std 1003.1-2001 does not specify behavior of concurrent writes to a file from multiple processes. Applications should use some form of concurrency control.

\textbf{FUTURE DIRECTIONS}
None.

\textbf{SEE ALSO}
\texttt{chmod()}, \texttt{creat()}, \texttt{dup()}, \texttt{fcntl()}, \texttt{getrlimit()}, \texttt{lseek()}, \texttt{open()}, \texttt{pipe()}, \texttt{ulimit()}, \texttt{write()}, the Base Definitions volume of IEEE Std 1003.1-2001, \texttt{<limits.h>}, \texttt{<stropts.h>}, \texttt{<sys/uio.h>}, \texttt{<unistd.h>}

\textbf{CHANGE HISTORY}
First released in Issue 1. Derived from Issue 1 of the SVID.

\textbf{Issue 5}
The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

The \texttt{pwrite()} function is added.

\textbf{Issue 6}
The DESCRIPTION states that the \texttt{write()} function does not block the thread. Previously this said “process” rather than “thread”.

The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:
• The DESCRIPTION now states that if `write()` is interrupted by a signal after it has successfully written some data, it returns the number of bytes written. In the POSIX.1-1988 standard, it was optional whether `write()` returned the number of bytes written, or whether it returned −1 with `errno` set to [EINTR]. This is a FIPS requirement.

• The following changes are made to support large files:
  
  — For regular files, no data transfer occurs past the offset maximum established in the open file description associated with the `fildes`.
  
  — A second [EFBIG] error condition is added.

• The [EIO] error condition is added.

• The [EPIPE] error condition is added for when a pipe has only one end open.

• The [ENXIO] optional error condition is added.

Text referring to sockets is added to the DESCRIPTION.

The following changes were made to align with the IEEE P1003.1a draft standard:

• The effect of reading zero bytes is clarified.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that `write()` results are unspecified for typed memory objects.

The following error conditions are added for operations on sockets: [EAGAIN], [EWOULDBLOCK], [ECONNRESET], [ENOTCONN], and [EPIPE].

The [EIO] error is changed to “may fail”.

The [ENOBUFS] error is added for sockets.

The following error conditions are added for operations on sockets: [EACCES], [ENETDOWN], and [ENETUNREACH].

The `writev()` function is split out into a separate reference page.
NAME
writev — write a vector

SYNOPSIS
XSI
#include <sys/uio.h>
ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);

DESCRIPTION
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>.

Each iovec entry specifies the base address and length of an area in memory from which data should be written. The writev() function shall always write a complete area before proceeding to the next.

If fildes refers to a regular file and all of the iov_len members in the array pointed to by iov are 0, writev() shall return 0 and have no other effect. For other file types, the behavior is unspecified.

If the sum of the iov_len values is greater than [SSIZE_MAX], the operation shall fail and no data shall be transferred.

RETURN VALUE
Upon successful completion, writev() shall return the number of bytes actually written. Otherwise, it shall return a value of −1, the file-pointer shall remain unchanged, and errno shall be set to indicate an error.

ERRORS
Refer to write().

In addition, the writev() function shall fail if:

[EINVAL] The sum of the iov_len values in the iov array would overflow an ssize_t.

The writev() function may fail and set errno to:

[EINVAL] The iovcnt argument was less than or equal to 0, or greater than [IOV_MAX].

EXAMPLES
Writing Data from an Array
The following example writes data from the buffers specified by members of the iov array to the file associated with the file descriptor fd.

#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
...
ssize_t bytes_written;
int fd;
char *buf0 = "short string\n";
char *buf1 = "This is a longer string\n";
char *buf2 = "This is the longest string in this example\n";
int iovcnt;
struct iovec iov[3];
writev()  

iov[0].iov_base = buf0;
iov[0].iov_len = strlen(buf0);
iov[1].iov_base = buf1;
iov[1].iov_len = strlen(buf1);
iov[2].iov_base = buf2;
iov[2].iov_len = strlen(buf2);
...
iovcnt = sizeof(iov) / sizeof(struct iovec);
bytes_written = writev(fd, iov, iovcnt);
...

APPLICATION USAGE
None.

RATIONALE
Refer to write().

FUTURE DIRECTIONS
None.

SEE ALSO
readv(), write(), the Base Definitions volume of IEEE Std 1003.1-2001, <limits.h>, <sys/uio.h>

CHANGE HISTORY
First released in Issue 4, Version 2.

Issue 6
Split out from the write() reference page.
NAME
wscanf — convert formatted wide-character input

SYNOPSIS
#include <stdio.h>
#include <wchar.h>
int wscanf(const wchar_t *restrict format, ...);

DESCRIPTION
Refer to fwscanf().
NAME
y0, y1, yn — Bessel functions of the second kind

SYNOPSIS
#include <math.h>

double y0(double x);
double y1(double x);
double yn(int n, double x);

DESCRIPTION
The y0(), y1(), and yn() functions shall compute Bessel functions of x of the second kind of
orders 0, 1, and n, respectively.
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.

RETURN VALUE
Upon successful completion, these functions shall return the relevant Bessel value of x of the
second kind.
If x is NaN, NaN shall be returned.
If the x argument to these functions is negative, –HUGE_VAL or NaN shall be returned, and a
domain error may occur.
If x is 0.0, –HUGE_VAL shall be returned and a range error may occur.
If the correct result would cause underflow, 0.0 shall be returned and a range error may occur.
If the correct result would cause overflow, –HUGE_VAL or 0.0 shall be returned and a range
error may occur.

ERRORS
These functions may fail if:

Domain Error The value of x is negative.
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.

Range Error The value of x is 0.0, or the correct result would cause overflow.
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.

Range Error The value of x is too large in magnitude, or the correct result would cause underflow.
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.
EXAMPLES
None.

APPLICATION USAGE
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.

RATIONALE
None.

FUTURE DIRECTIONS
None.

SEE ALSO
gescalrecept(), fetestexcept(), isnan(), j0(), the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.18, Treatment of Error Conditions for Mathematical Functions, <math.h>

CHANGE HISTORY
First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5
The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6
The DESCRIPTION is updated to avoid use of the term “must” for application requirements.
The RETURN VALUE and ERRORS sections are reworked for alignment of the error handling with the ISO/IEC 9899: 1999 standard.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SC_RTSIG_MAX</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_SAVED_IDS</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_SEMAPHORES</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_SEM_NSEMS_MAX</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_SEM_VALUE_MAX</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_SHARED_MEMORY_OBJECTS</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_SHELL</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_SIGQUEUE_MAX</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_SPAWN</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_SPIN_LOCKS</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_SPODARIC_SERVER</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_STREAM_MAX</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_SYMLOOP_MAX</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_SYNCHRONIZED_IO</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_THREADS</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_THREAD_ATTR_STACKADDR</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_THREAD_ATTR_STACKSIZE</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_THREAD_CPUTIME</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_THREAD_DESTRUCTOR_ITERATIONS</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_THREAD_KEYS_MAX</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_THREAD_PRIORITY_SCHEDULING</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_THREAD_PRIO_INHERIT</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_THREAD_PRIO_PROTECT</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_THREAD_PROCESS_SHARED</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_THREAD_SAFE_FUNCTIONS</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_THREAD_SPODARIC_SERVER</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_THREAD_STACK_MIN</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_THREAD_THREADS_MAX</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_TIMER_MAX</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_TRACE</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_TRACE_EVENT_FILTER</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_TRACE_INHERIT</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_TRACE_LOG</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_TTY_NAME_MAX</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_TYPED_MEMORY_OBJECTS</td>
<td>1467</td>
</tr>
<tr>
<td>_SC_TZNAME_MAX</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_V6_ILP32_OFF32</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_V6_ILP64_OFF64</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_V6_LP64_OFF64</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_V6_LPBIG_OFF64</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_VERSION</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_XBS5_ILP32_OFF32</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_XBS5_ILP32_OFFBIG</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_XBS5_LP64_OFF64</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_XBS5_LPBIG_OFF64</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_XOPEN_CRYPT</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_XOPEN_ENH_I18N</td>
<td>1468</td>
</tr>
<tr>
<td>_SC_XOPEN_LEGACY</td>
<td>1469</td>
</tr>
<tr>
<td>_SC_XOPEN_REALTIME</td>
<td>1469</td>
</tr>
<tr>
<td>_SC_XOPEN_REALTIME_THREADS</td>
<td>1469</td>
</tr>
<tr>
<td>_SC_XOPEN_SHM</td>
<td>1469</td>
</tr>
<tr>
<td>_SC_XOPEN_UNIX</td>
<td>1469</td>
</tr>
<tr>
<td>_SC_XOPEN_VERSION</td>
<td>1469</td>
</tr>
<tr>
<td>_SC_XOPEN_XCU_VERSION</td>
<td>1469</td>
</tr>
<tr>
<td>_setjmp</td>
<td>86</td>
</tr>
<tr>
<td>_t</td>
<td>17</td>
</tr>
<tr>
<td>_TIME</td>
<td>17</td>
</tr>
<tr>
<td>_tolower</td>
<td>88</td>
</tr>
<tr>
<td>_toupper</td>
<td>89</td>
</tr>
<tr>
<td>_XBS5_ILP32_OFF32</td>
<td>1468</td>
</tr>
<tr>
<td>_XBS5_ILP32_OFFBIG</td>
<td>1468</td>
</tr>
<tr>
<td>_XBS5_LP64_OFF64</td>
<td>1468</td>
</tr>
<tr>
<td>_XBS5_LPBIG_OFF64</td>
<td>1468</td>
</tr>
<tr>
<td>_XOPEN_CRYPT</td>
<td>1468</td>
</tr>
<tr>
<td>_XOPEN_ENH_I18N</td>
<td>1468</td>
</tr>
<tr>
<td>_XOPEN_LEGACY</td>
<td>1469</td>
</tr>
<tr>
<td>_XOPEN_REALTIME</td>
<td>335, 1469</td>
</tr>
<tr>
<td>_XOPEN_REALTIME_THREADS</td>
<td>1469</td>
</tr>
<tr>
<td>_XOPEN_SHM</td>
<td>1469</td>
</tr>
<tr>
<td>_XOPEN_SOURCE</td>
<td>14</td>
</tr>
<tr>
<td>_XOPEN_UNIX</td>
<td>1469</td>
</tr>
<tr>
<td>_XOPEN_VERSION</td>
<td>1469</td>
</tr>
<tr>
<td>_XOPEN_XCU_VERSION</td>
<td>1469</td>
</tr>
<tr>
<td>_a64l</td>
<td>90</td>
</tr>
<tr>
<td>_ABDAY_1</td>
<td>831</td>
</tr>
<tr>
<td>_abort</td>
<td>92</td>
</tr>
<tr>
<td>_abs</td>
<td>93</td>
</tr>
<tr>
<td>_accept</td>
<td>94</td>
</tr>
<tr>
<td>_access</td>
<td>96</td>
</tr>
<tr>
<td>acos</td>
<td>99</td>
</tr>
<tr>
<td>acosf</td>
<td>99</td>
</tr>
<tr>
<td>acosh</td>
<td>101</td>
</tr>
<tr>
<td>acoshf</td>
<td>101</td>
</tr>
<tr>
<td>acoshl</td>
<td>101</td>
</tr>
<tr>
<td>acosl</td>
<td>99, 103</td>
</tr>
<tr>
<td>_ACTION</td>
<td>581</td>
</tr>
<tr>
<td>_addrinfo</td>
<td>424</td>
</tr>
<tr>
<td>_address info</td>
<td>424</td>
</tr>
<tr>
<td>_address string</td>
<td>424</td>
</tr>
<tr>
<td>ADV</td>
<td>3</td>
</tr>
<tr>
<td>ADVANCED REALTIME</td>
<td>194, 198, 797-798, 861</td>
</tr>
<tr>
<td>...</td>
<td>863, 865, 867, 869, 872, 880, 883</td>
</tr>
<tr>
<td>...</td>
<td>885-887, 889, 891, 893, 895, 897</td>
</tr>
<tr>
<td>...</td>
<td>899, 901-908, 961, 963, 1039</td>
</tr>
<tr>
<td>...</td>
<td>1084, 1253</td>
</tr>
<tr>
<td>ADVANCED REALTIME THREADS</td>
<td>1008, 1010</td>
</tr>
<tr>
<td>...</td>
<td>1012, 1014, 1016-1017, 1056, 1137</td>
</tr>
<tr>
<td>...</td>
<td>1139, 1141</td>
</tr>
<tr>
<td>AF</td>
<td>18</td>
</tr>
</tbody>
</table>
Index

cacoshf() ...................................................................155
cacoshl() ...................................................................155
cacos() ...................................................................154, 156
calloc() ...................................................................157
can........................................................................1
cancel-safe ..................................................................1129
cancelability state..................................................54, 1063, 1129
cancelability type ..................................................1063, 1129
cancelation cleanup handler ...................................1023, 1034
...............................................................................1052, 1067
cancelation points ......................................................55
canceling execution of a thread ..................................1018
canonical name ..........................................................425
carg() ....................................................................159
cargf() ....................................................................159
cargl() ....................................................................159
casin() ....................................................................160
casinf() ....................................................................160
casinh() ....................................................................161
casinhf() ....................................................................161
casinhl() ....................................................................161
casinl() ....................................................................160, 162
catan() ....................................................................163
catand() ....................................................................163
catanh() ....................................................................164
catanhf() ....................................................................164
catanhl() ....................................................................164
cataml() ....................................................................163, 165
catclose() ..................................................................166
catgets() ....................................................................167
catopen() ....................................................................169
CBAUD ........................................................................18
cbrt() .....................................................................171
cbrtf() .....................................................................171
cbrtl() .....................................................................171
ccos() .....................................................................172
ccosf() .....................................................................172
cosh() .....................................................................173
coshf() .....................................................................173
coshl() .....................................................................173
ccosl() .....................................................................172, 174
CD .............................................................................4
ceil() .....................................................................175
ceilf() .....................................................................175
ceill() .....................................................................175
cexp() .....................................................................177
cexpf() .....................................................................177
cexpl() .....................................................................177
cfsetospeed() ..........................................................182
cfgetispeed() ..........................................................155
cfgetospeed() ..........................................................180
cfsetispeed() ..........................................................181
change current working directory ................................184
change file modes......................................................187
change owner and group of file ..................................189
CHAR_MAX ..................................................................694, 696
chdir() .....................................................................183
CHILD_MAX ................................................................393, 1466
chmod() .....................................................................185
chown() .....................................................................188
cimag() .....................................................................191
cimagf() .....................................................................191
cimagl() .....................................................................191
CLD ........................................................................18
clearerr() ...................................................................192
clock() ....................................................................193
clock() .....................................................................193
clock-resolution attribute ........................................76, 911
CLOCKS_PER_SEC ..................................................193
CLOCK ........................................................................17
clock_gettime() ........................................................195
clock_getres() ........................................................194
clock_gettime() .......................................................195
CLOCK_MONOTONIC ...............................................49, 199
clock_gettime() ........................................................198
CLOCK_PROCESS_CPUTIME_ID ....................................50
CLOCK_REALTIME ....................................................49, 195, 199
...............................................................................820, 1084, 1253, 1514
clock_settime() ........................................................195, 201
CLOCK_THREAD_CPUTIME_ID ....................................50
clog() .....................................................................202
clogf() .....................................................................202
clogl() .....................................................................202
close a file .................................................................205
close() .....................................................................203
closedir() ...................................................................206
closelog() ...................................................................208
cmsg .................................................................1582, 1583, 1594, 1600
CMS_ID .....................................................................18
COLL_WEIGHTS_MAX ............................................1466
command interpreter ................................................1585
portable .....................................................................1585
compare thread IDs ..................................................1051
compilation environment ..............................................13
condition variable initialization attributes ..................1037
conforming application ............................................1383
conforming application, strictly ................................118, 300
confstr() ...................................................................212
conj() .....................................................................215
cimag() .....................................................................191
cimagf() .....................................................................191
cimagl() .....................................................................191
CLD ........................................................................18
clearerr() ...................................................................192
clock() ....................................................................193
clock() ....................................................................193
clock-resolution attribute ........................................76, 911
CLOCKS_PER_SEC ..................................................193
CLOCK ........................................................................17
clock_gettime() ........................................................195
clock_getres() ........................................................194
clock_gettime() .......................................................195
CLOCK_MONOTONIC ...............................................49, 199
clock_gettime() ........................................................198
CLOCK_PROCESS_CPUTIME_ID ....................................50
CLOCK_REALTIME ....................................................49, 195, 199
...............................................................................820, 1084, 1253, 1514
clock_settime() ........................................................195, 201
CLOCK_THREAD_CPUTIME_ID ....................................50
clog() .....................................................................202
clogf() .....................................................................202
clogl() .....................................................................202
close a file .................................................................205
close() .....................................................................203
closedir() ...................................................................206
closelog() ...................................................................208
cmsg .................................................................1582, 1583, 1594, 1600
CMS_ID .....................................................................18
COLL_WEIGHTS_MAX ............................................1466
Index

control data ................................................................. 38
control-normal .............................................................. 1168
conversion descriptor .................................................. 295, 300, 587-590
conversion specification ................................................. 402, 433, 466
modified ........................................................................ 1424
conversion specifier modified ........................................ 1435
copysign() .................................................................. 219
copysignf() .................................................................. 219
copysignl() .................................................................. 219
core ............................................................................ 1586
core file .......................................................................... 307
cos() ........................................................................... 220
cosf() ........................................................................... 220
cosh() ........................................................................... 222
coshf() ......................................................................... 222
coshl() ......................................................................... 222
cosl() ............................................................................ 220, 224
covet channel ................................................................ 666
cpom() ........................................................................... 225
cpomf() ........................................................................... 225
cpowl() ........................................................................... 225
cproj() ........................................................................... 226
cprojf() ........................................................................... 226
cprofl() ........................................................................... 226
CPT ................................................................................ 4
creal() ........................................................................... 227
crealf() ........................................................................... 227
creall() ........................................................................... 227
creat() ............................................................................ 228
create a per-process timer ............................................ 1515
create an interprocess channel ................................... 853
create session and set process group ID ..................... 1311
creation-time attribute ................................................ 76, 911
CRYPT ........................................................................... 230, 268, 1292
crypt() ........................................................................... 230
CS .................................................................................... 4
csin() ............................................................................ 232
csinf() ............................................................................ 232
csinh() ........................................................................... 233
csinhf() ........................................................................... 233
csinhl() ........................................................................... 233
csinf() ............................................................................ 232
csinp() ............................................................................ 234
csqrt() ........................................................................... 235
csqrtf() ........................................................................... 235
csqrtl() ........................................................................... 235
ctan() ............................................................................. 236
ctan() ............................................................................. 236
ctanf() ............................................................................ 236
ctanhl() ........................................................................... 237
ctanhf() ........................................................................... 237
ctanhl() ........................................................................... 237
ctanl() ............................................................................ 256, 238
ctermid() ........................................................................ 239
cftime() ........................................................................... 241
cftime_r() ....................................................................... 241
CXX .............................................................................. 4
c_ .................................................................................... 17
data key creation ........................................................ 1067
data messages ............................................................ 38
data type .......................................................................... 50
DATEMSK ........................................................................ 493
daylight .......................................................................... 243, 1542
DBL_MANT_DIG .......................................................... 175, 375
DBL_MAX_EXP .................................................................. 175, 375
DBM ............................................................................. 244-245
dbms ............................................................................... 16
DBM ............................................................................. 18
dbm_clearerr() ............................................................ 244
dbm_close() .................................................................... 244
dbm_delete() .................................................................... 244
dbm_error() ..................................................................... 244
dbm_fetch() ..................................................................... 244
dbm_firstkey() .............................................................. 244
DBM_INSERT ................................................................... 244
dbm_nextkey() .............................................................. 244
dbm_open() ....................................................................... 244
DBM_REPLACE ................................................................ 244
dbm_store() ..................................................................... 244
DEAD_PROCESS ................................................................ 282-283
DEFECCHO ....................................................................... 18
deferred cancelability ................................................ 1063
delay process execution .............................................. 1382
DELAYTIMER_MAX ......................................................... 1466, 1519
dependency ordering .................................................... 257
descriptive name ........................................................ 424
destroying a mutex ....................................................... 1073
destroying condition variables ................................... 1030
destructor functions ..................................................... 1067
detaching a thread ....................................................... 1049
difftime() ....................................................................... 248
DIRECTORY ....................................................................... 80, 206, 842, 1174, 1176, 1214, 1240, 1505
directive .......................................................................... 402, 433, 466, 475, 1434
directory operations .................................................... 843
dirent structure ............................................................ 843
dirname() ......................................................................... 249
div() ............................................................................... 251
dlclose() .......................................................................... 252
dlerror() .......................................................................... 254
# Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dlopen()</td>
<td>256</td>
</tr>
<tr>
<td>dlsym()</td>
<td>259</td>
</tr>
<tr>
<td>dot...</td>
<td>843, 1211</td>
</tr>
<tr>
<td>dot-dot...</td>
<td>843, 1211</td>
</tr>
<tr>
<td>drand48()</td>
<td>261</td>
</tr>
<tr>
<td>dup()</td>
<td>263</td>
</tr>
<tr>
<td>dup2()</td>
<td>263</td>
</tr>
<tr>
<td>dynamic package initialization</td>
<td>1105</td>
</tr>
<tr>
<td>d_...</td>
<td>16</td>
</tr>
<tr>
<td>E2BIG</td>
<td>22</td>
</tr>
<tr>
<td>EACCES</td>
<td>22</td>
</tr>
<tr>
<td>EADDRINUSE</td>
<td>22</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>22</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>22</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>22, 28</td>
</tr>
<tr>
<td>EALREADY</td>
<td>22</td>
</tr>
<tr>
<td>EBAADF</td>
<td>22</td>
</tr>
<tr>
<td>EBADMSG</td>
<td>22</td>
</tr>
<tr>
<td>EBUSY</td>
<td>23</td>
</tr>
<tr>
<td>ECANCELED</td>
<td>23</td>
</tr>
<tr>
<td>ECHILD</td>
<td>23</td>
</tr>
<tr>
<td>ECHOSTCTL</td>
<td>18</td>
</tr>
<tr>
<td>ECHOKE</td>
<td>18</td>
</tr>
<tr>
<td>ECHOPRT</td>
<td>18</td>
</tr>
<tr>
<td>ECONNABORTED</td>
<td>23</td>
</tr>
<tr>
<td>ECONNREFUSED</td>
<td>23</td>
</tr>
<tr>
<td>ECONNECT</td>
<td>23</td>
</tr>
<tr>
<td>ECVT()</td>
<td>266</td>
</tr>
<tr>
<td>EDEADLK</td>
<td>23</td>
</tr>
<tr>
<td>EDESTADDRREQ</td>
<td>23</td>
</tr>
<tr>
<td>EDOM</td>
<td>23</td>
</tr>
<tr>
<td>EDQUOT</td>
<td>23</td>
</tr>
<tr>
<td>EEXIST</td>
<td>23</td>
</tr>
<tr>
<td>EFAULT</td>
<td>23</td>
</tr>
<tr>
<td>EFBIG</td>
<td>23</td>
</tr>
<tr>
<td>effective group ID</td>
<td>189, 302, 511</td>
</tr>
<tr>
<td>effective user ID</td>
<td>97, 302, 666</td>
</tr>
<tr>
<td>EHOLD</td>
<td>24</td>
</tr>
<tr>
<td>EIDRM</td>
<td>24</td>
</tr>
<tr>
<td>Eighth Edition UNIX</td>
<td>1654</td>
</tr>
<tr>
<td>EILSEQ</td>
<td>24, 38</td>
</tr>
<tr>
<td>EINPROGRESS</td>
<td>24, 43</td>
</tr>
<tr>
<td>EINTR</td>
<td>24, 57, 930</td>
</tr>
<tr>
<td>EINVAL</td>
<td>24, 917, 930</td>
</tr>
<tr>
<td>EIO</td>
<td>24</td>
</tr>
<tr>
<td>EISCONN</td>
<td>24</td>
</tr>
<tr>
<td>EISDIR</td>
<td>24</td>
</tr>
<tr>
<td>ELOOP</td>
<td>24</td>
</tr>
<tr>
<td>ELSIZE</td>
<td>723</td>
</tr>
<tr>
<td>EMPFILE</td>
<td>24</td>
</tr>
<tr>
<td>EMLINK</td>
<td>24</td>
</tr>
<tr>
<td>ENFILE</td>
<td>25</td>
</tr>
<tr>
<td>ENETDOWN</td>
<td>25</td>
</tr>
<tr>
<td>ENETRESET</td>
<td>25</td>
</tr>
<tr>
<td>ENETUNREACH</td>
<td>25</td>
</tr>
<tr>
<td>ENFILE</td>
<td>25</td>
</tr>
<tr>
<td>ENOBUSFS</td>
<td>25</td>
</tr>
<tr>
<td>ENODATA</td>
<td>25</td>
</tr>
<tr>
<td>ENODEV</td>
<td>25</td>
</tr>
<tr>
<td>ENOENT</td>
<td>25</td>
</tr>
<tr>
<td>ENOEXEC</td>
<td>25</td>
</tr>
<tr>
<td>ENOLCK</td>
<td>25</td>
</tr>
<tr>
<td>ENOLINK</td>
<td>25</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>25</td>
</tr>
<tr>
<td>ENOMSG</td>
<td>25</td>
</tr>
<tr>
<td>ENOPROTOOPT</td>
<td>26</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>26</td>
</tr>
<tr>
<td>ENOSR</td>
<td>26</td>
</tr>
<tr>
<td>ENOTOTY</td>
<td>26</td>
</tr>
<tr>
<td>ENTRY</td>
<td>.581</td>
</tr>
<tr>
<td>environ</td>
<td>.285, 301</td>
</tr>
<tr>
<td>envp</td>
<td>.301</td>
</tr>
<tr>
<td>EXNIO</td>
<td>.26</td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td>.26</td>
</tr>
<tr>
<td>EOVERFLOW</td>
<td>.26</td>
</tr>
<tr>
<td>EPERM</td>
<td>.27</td>
</tr>
<tr>
<td>EPIPE</td>
<td>.27</td>
</tr>
<tr>
<td>EPROTO</td>
<td>.27</td>
</tr>
<tr>
<td>EPROTONOSUPPORT</td>
<td>.27</td>
</tr>
<tr>
<td>EPROTOTYPE</td>
<td>.27</td>
</tr>
<tr>
<td>erand48()</td>
<td>.261, 286</td>
</tr>
<tr>
<td>erf()</td>
<td>.287</td>
</tr>
<tr>
<td>erfc()</td>
<td>.289</td>
</tr>
<tr>
<td>Feature</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>erfci()</td>
<td>289</td>
</tr>
<tr>
<td>erfc()</td>
<td>289</td>
</tr>
<tr>
<td>erf()</td>
<td>287, 291</td>
</tr>
<tr>
<td>erf()</td>
<td>287, 291</td>
</tr>
<tr>
<td>EROFS</td>
<td>27</td>
</tr>
<tr>
<td>errno</td>
<td>292</td>
</tr>
<tr>
<td>error descriptions</td>
<td>481</td>
</tr>
<tr>
<td>error numbers</td>
<td>21</td>
</tr>
<tr>
<td>additional</td>
<td>28</td>
</tr>
<tr>
<td>ESPIPE</td>
<td>27</td>
</tr>
<tr>
<td>ESRC</td>
<td>27</td>
</tr>
<tr>
<td>EST5EDT</td>
<td>1542</td>
</tr>
<tr>
<td>establishing cancelation handlers</td>
<td>1023</td>
</tr>
<tr>
<td>ESTALE</td>
<td>27</td>
</tr>
<tr>
<td>ETIME</td>
<td>27</td>
</tr>
<tr>
<td>ETIMEDOUT</td>
<td>27</td>
</tr>
<tr>
<td>ETXTBSY</td>
<td>27</td>
</tr>
<tr>
<td>EWOULDBLOCK</td>
<td>28</td>
</tr>
<tr>
<td>examine and change blocked signals</td>
<td>1136</td>
</tr>
<tr>
<td>examine and change signal action</td>
<td>1342</td>
</tr>
<tr>
<td>EXDEV</td>
<td>28</td>
</tr>
<tr>
<td>exec</td>
<td>294</td>
</tr>
<tr>
<td>exec family</td>
<td>300</td>
</tr>
<tr>
<td>exec of shell scripts</td>
<td>300</td>
</tr>
<tr>
<td>exec family...97, 205, 331, 372, 395, 976, 1300, 1585</td>
<td>294</td>
</tr>
<tr>
<td>exec</td>
<td>294</td>
</tr>
<tr>
<td>execle</td>
<td>294</td>
</tr>
<tr>
<td>execle...</td>
<td>294</td>
</tr>
<tr>
<td>execle</td>
<td>294</td>
</tr>
<tr>
<td>execute a file</td>
<td>294</td>
</tr>
<tr>
<td>execute a file...</td>
<td>294</td>
</tr>
<tr>
<td>execution time monitoring</td>
<td>50</td>
</tr>
<tr>
<td>execv</td>
<td>294</td>
</tr>
<tr>
<td>execute</td>
<td>294</td>
</tr>
<tr>
<td>execvp</td>
<td>294</td>
</tr>
<tr>
<td>exit()</td>
<td>305</td>
</tr>
<tr>
<td>EXIT_FAILURE</td>
<td>305</td>
</tr>
<tr>
<td>EXIT_SUCCESS</td>
<td>305, 308</td>
</tr>
<tr>
<td>exp()</td>
<td>310</td>
</tr>
<tr>
<td>exp2()</td>
<td>312</td>
</tr>
<tr>
<td>exp2f()</td>
<td>312</td>
</tr>
<tr>
<td>exp2l()</td>
<td>312</td>
</tr>
<tr>
<td>expf()</td>
<td>310</td>
</tr>
<tr>
<td>expl()</td>
<td>310</td>
</tr>
<tr>
<td>expm1f()</td>
<td>314</td>
</tr>
<tr>
<td>expm1f()</td>
<td>314</td>
</tr>
<tr>
<td>EXPR_NEST_MAX</td>
<td>1466</td>
</tr>
<tr>
<td>EXTA</td>
<td>18</td>
</tr>
<tr>
<td>EXTB</td>
<td>18</td>
</tr>
<tr>
<td>extension</td>
<td>4</td>
</tr>
<tr>
<td>CX</td>
<td>4</td>
</tr>
<tr>
<td>OH</td>
<td>6</td>
</tr>
<tr>
<td>XSI</td>
<td>10</td>
</tr>
<tr>
<td>extensions to setlocale</td>
<td>1294</td>
</tr>
<tr>
<td>fabs()</td>
<td>316</td>
</tr>
<tr>
<td>fabsf()</td>
<td>316</td>
</tr>
<tr>
<td>fabsl()</td>
<td>316</td>
</tr>
<tr>
<td>fattach()</td>
<td>317</td>
</tr>
<tr>
<td>fcchal()</td>
<td>320</td>
</tr>
<tr>
<td>fchmod()</td>
<td>321</td>
</tr>
<tr>
<td>fchown()</td>
<td>323</td>
</tr>
<tr>
<td>fclose()</td>
<td>325</td>
</tr>
<tr>
<td>fcntl()</td>
<td>327</td>
</tr>
<tr>
<td>fctv()</td>
<td>266, 334</td>
</tr>
<tr>
<td>FD</td>
<td>4</td>
</tr>
<tr>
<td>fdatasync()</td>
<td>335</td>
</tr>
<tr>
<td>fdetach()</td>
<td>336</td>
</tr>
<tr>
<td>fdim()</td>
<td>338</td>
</tr>
<tr>
<td>fdimf()</td>
<td>338</td>
</tr>
<tr>
<td>fdiml()</td>
<td>338</td>
</tr>
<tr>
<td>fdopen()</td>
<td>340</td>
</tr>
<tr>
<td>fds_</td>
<td>16</td>
</tr>
<tr>
<td>fd_</td>
<td>16</td>
</tr>
<tr>
<td>FD_CLOEXEC</td>
<td>16, 18</td>
</tr>
<tr>
<td>feature test macro</td>
<td>835, 843, 853, 873, 880, 1323</td>
</tr>
<tr>
<td>_POSIX_C_SOURCE</td>
<td>13, 486</td>
</tr>
<tr>
<td>_XOPEN_SOURCE</td>
<td>14</td>
</tr>
<tr>
<td>fclearexcept()</td>
<td>342</td>
</tr>
<tr>
<td>fegetenv()</td>
<td>343</td>
</tr>
<tr>
<td>fegetexceptflag()</td>
<td>344</td>
</tr>
<tr>
<td>fegetround()</td>
<td>345</td>
</tr>
<tr>
<td>feholdexcept()</td>
<td>347</td>
</tr>
<tr>
<td>feof()</td>
<td>348</td>
</tr>
<tr>
<td>feraiseexcept()</td>
<td>349</td>
</tr>
<tr>
<td>ferror()</td>
<td>350</td>
</tr>
<tr>
<td>fesetenv()</td>
<td>343</td>
</tr>
<tr>
<td>fesetenv()</td>
<td>351</td>
</tr>
<tr>
<td>fesetexceptflag()</td>
<td>344, 352</td>
</tr>
<tr>
<td>fesetround()</td>
<td>345, 353</td>
</tr>
<tr>
<td>fetestexcept()</td>
<td>354</td>
</tr>
<tr>
<td>feupdateenv()</td>
<td>356</td>
</tr>
<tr>
<td>fflush()</td>
<td>358</td>
</tr>
<tr>
<td>ffs()</td>
<td>361</td>
</tr>
<tr>
<td>fgetc()</td>
<td>362</td>
</tr>
<tr>
<td>fgetpos()</td>
<td>364</td>
</tr>
<tr>
<td>fgets()</td>
<td>366</td>
</tr>
<tr>
<td>fgetwc()</td>
<td>368</td>
</tr>
</tbody>
</table>
Index

getsockopt() .......................................................... 560
getsockopt() .......................................................... 563
gettimeofday() ...................................................... 567
gettimeofday() ...................................................... 568
gettimeofday() ...................................................... 569
gettimeofday() ...................................................... 569
gettimeofday() ...................................................... 570
gettimeofday() ...................................................... 571
gettimeofday() ...................................................... 492, 572
GETVAL ............................................................. 1260-1261
glob() .................................................................... 573
globfree() ................................................................ 573
GLOB ........................................................................ 18
GLOB_ constants
   error returns of glob ........................................... 575
   used in glob ........................................................ 573
GLOB_ABORTED ...................................................... 575
GLOB_APPEND ........................................................ 573-574
GLOB_DOOFS ........................................................ 573-574
GLOB_ERR .......................................................... 573, 575
GLOB_MARK .......................................................... 574
GLOB_NOCHECK .................................................... 574-575
GLOB_NOESCAPE .................................................. 574
GLOB_NOMATCH .................................................... 575
GLOB_NOSORT ....................................................... 574
GLOB_NOSPACE .................................................... 575
gl............................................................................. 16
GMTO ................................................................. 1542
gmtime() .................................................................... 577
gmtime_r() .............................................................. 577
grantpt() .................................................................. 579
granularity of clock ............................................... 454
HALT ........................................................................ 384
hcreate() .................................................................. 581
hdestr() ..................................................................... 581
high resolution sleep .............................................. 820
host name .................................................................. 424
htonl() ...................................................................... 584
htons() ...................................................................... 584
HUGE_VAL ............................................................. 133, 175, 222, 310, 312
................................................................. 314, 338, 375, 585, 674, 678, 705
................................................................. 709, 711, 822, 824, 966, 1216
................................................................. 1221, 1225, 1379, 1442, 1478
................................................................. 1509, 1612
HUGE_VALF .......................................................... 375, 1216, 1442, 1612
HUGE_VALL .......................................................... 375, 1216, 1442, 1612
hypot() ...................................................................... 585
hypot() ...................................................................... 585
hypot() ...................................................................... 585
hypot() ...................................................................... 585
hypot() ...................................................................... 585
hypot() ...................................................................... 585

h_................................................................. 16
h_errno ............................................................... 580
iconv() ................................................................. 587
iconv_close() .......................................................... 589
iconv_open() .......................................................... 590
ic ........................................................................... 16
IEEE Std 754-1985 ................................................. 3
IEEE Std 854-1987 ................................................. 3
if_ ............................................................................ 16
ifra_ ........................................................................ 16
ifrui ......................................................................... 16
if_ ............................................................................ 16
IF ............................................................................ 18
if_freenameindex() ............................................... 592
if_indextoname() ..................................................... 593
if_nameindex() ....................................................... 594
if_nameindex() ....................................................... 595
ILL_ ......................................................................... 18
ilogb() .................................................................... 596
ilogbf() .................................................................... 596
ilogb() .................................................................... 596
imaxabs() .............................................................. 598
imaxdiv() .............................................................. 599
implementation-defined ........................................ 1
IMPLINK_ ................................................................ 18
in6_ ........................................................................ 16
Inf ......................................................................... 122
INF ......................................................................... 405, 469
index() .................................................................... 600
inet_ ......................................................................... 16
inet_ addr() ............................................................ 601
inet_سلاح() ........................................................... 601
inet_pton() ............................................................. 603
inet_pton() ............................................................. 603
INFITY ................................................................. 405, 469
INFO ......................................................................... 384
infu ........................................................................... 16
inheritance attribute ............................................ 76, 913
init ......................................................................... 308, 666
initialize a named semaphore ................................ 1249
initialize an unnamed semaphore ........................... 1246
initializing a mutex .................................................. 1073
initializing condition variables ................................ 1030
initstate() ............................................................... 605
INIT_PROCESS ....................................................... 282-283
input and output rationale ...................................... 1170
insq() ..................................................................... 607
INT ........................................................................... 18
international environment ...................................... 1294
Internet Protocols ...................................................... 66
INT_MAX................................................................. 596
INT_MIN ................................................................... 93
in_ ........................................................................... 16
IN_ ........................................................................... 18
ioctl() .................................................................... 610
iov_ ......................................................................... 17
IOV_ ......................................................................... 18
IOV_MAX................................................................... 1181, 1466, 1657
IP6 ............................................................................. 5
IPＣ_ ............................................................................. 39, 803, 805, 808, 810, 1265, 1269, 1333, 1335
ipc_ ............................................................................. 16
IPC_ ............................................................................. 18
IPC_ constants
  used in semctl ....................................................... 1260
  used in shmctl ....................................................... 1331
IPC_CREAT ............................................................. 804, 1263, 1334
IPC_EXCL ............................................................... 804, 1263
IPC_NOWAIT ........................................................... 806-807, 809-810, 1266
IPC_PRIVATE ........................................................ 804, 1263, 1334
IPC_RMID ............................................................... 802, 1261, 1331
IPC_SET ............................................................... 802, 1261, 1331
IPC_STAT ............................................................... 802, 1260, 1331
IPPORT_ ................................................................. 18
IPPROTO_ .................................................................. 18
IPV6 ................................................................. 67
IPv4-compatible address ............................................. 68
IPv4-mapped address .................................................. 68
IPv6 ....................................................................... 67
  compatibility with IPv4 ............................................. 68
  interface identification .......................................... 68
  options .................................................................. 69
IPv6 address
  anycast .................................................................. 67
  loopback .............................................................. 68
  multicast .................................................................. 67
  unicast .................................................................. 67
  unspecified .......................................................... 68
IPV6_ ................................................................. 18
IPV6_JOIN_GROUP ............................................... 69
IPV6_LEAVE_GROUP ............................................... 69
IPV6_MULTICAST_HOPS ........................................... 69
IPV6_MULTICAST_IF .................................................. 69
IPV6_MULTICAST_LOOP ............................................ 69
IPV6_UNICAST_HOPS ............................................... 69
IPV6_V6ONLY ........................................................ 69
ip_ ........................................................................... 16
IP_ ........................................................................... 18
isalnum() ................................................................ 622
isalpha() ................................................................... 623
isascii() .................................................................. 624
isbclass() .................................................................. 625
isb convin( ) ................................................................ 626
isblank() .................................................................. 627
iscntrl() ................................................................... 628
isdigit() .................................................................... 629
isfinite() ................................................................... 630
isgraph() ................................................................... 631
isgreater() ................................................................ 632
isgreaterequal() ...................................................... 633
isinf() ....................................................................... 634
isless() ...................................................................... 635
islessequal() .......................................................... 636
islessgreater() ........................................................ 637
islower() .................................................................... 638
isnan() ....................................................................... 640
isnormal() .................................................................. 641
ISO C standard ......................................................... 3, 118, 300, 330
................................................................. 486, 725, 1164, 1211, 1294
................................................................. 1342, 1366, 1512
issprint() .................................................................... 642
isspace() .................................................................... 644
issup rem() .............................................................. 645
isunordered() .......................................................... 646
iswalnum() .............................................................. 647
iswalpha() ............................................................... 648
iswblank() ............................................................... 649
iscantrl() ................................................................. 650
iswctype() ............................................................... 651
iswctype() ............................................................... 652
iswctype() ............................................................... 653
iswctype() ............................................................... 654
iswctype() ............................................................... 655
iswctype() ............................................................... 656
iswctype() ............................................................... 657
iswctype() ............................................................... 658
iswctype() ............................................................... 659
iswctype() ............................................................... 660
iswctype() ............................................................... 661
ITIMER_PROF .......................................................... 517
ITIMER_REAL .......................................................... 517
ITIMER_VIRTUAL ....................................................... 517
it_ ............................................................................. 16-17
L ............................................................................ 18
I_ATMARK .............................................................. 616-617
I_CANPUT .............................................................. 617
I_CKBAND .............................................................. 617
I_FDINSERT .......................................................... 613
I_FIND ................................................................. 612
I_FLUSH ................................................................. 610
I_FLUSHBAND ......................................................... 611
I_GETBAND ........................................................... 617
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>log1pl()</td>
<td>709</td>
</tr>
<tr>
<td>log2()</td>
<td>711</td>
</tr>
<tr>
<td>log2f()</td>
<td>711</td>
</tr>
<tr>
<td>logb()</td>
<td>713</td>
</tr>
<tr>
<td>logbf()</td>
<td>713</td>
</tr>
<tr>
<td>logbl()</td>
<td>713</td>
</tr>
<tr>
<td>logf()</td>
<td>705, 715</td>
</tr>
<tr>
<td>login shell</td>
<td>300</td>
</tr>
<tr>
<td>LOGIN_NAME_MAX</td>
<td>519, 1466</td>
</tr>
<tr>
<td>LOGIN_PROCESS</td>
<td>282-283</td>
</tr>
<tr>
<td>logl</td>
<td>705, 715</td>
</tr>
<tr>
<td>LOG_</td>
<td>18</td>
</tr>
<tr>
<td>LOG_constants in syslog</td>
<td>208</td>
</tr>
<tr>
<td>LOG_ALERT</td>
<td>208</td>
</tr>
<tr>
<td>LOG_CONS</td>
<td>209</td>
</tr>
<tr>
<td>LOG_CRIT</td>
<td>208</td>
</tr>
<tr>
<td>LOG_DEBUG</td>
<td>208</td>
</tr>
<tr>
<td>LOG_EMERG</td>
<td>208</td>
</tr>
<tr>
<td>LOG_ERR</td>
<td>208</td>
</tr>
<tr>
<td>LOG_INFO</td>
<td>208</td>
</tr>
<tr>
<td>LOG_LOCAL</td>
<td>208</td>
</tr>
<tr>
<td>LOG_NDELAY</td>
<td>209</td>
</tr>
<tr>
<td>LOG_NOTICE</td>
<td>208</td>
</tr>
<tr>
<td>LOG_NOWAIT</td>
<td>209</td>
</tr>
<tr>
<td>LOG_ODELAY</td>
<td>209</td>
</tr>
<tr>
<td>LOG_PID</td>
<td>209</td>
</tr>
<tr>
<td>LOG_USER</td>
<td>208-209</td>
</tr>
<tr>
<td>LOG_WARNING</td>
<td>208</td>
</tr>
<tr>
<td>longjmp()</td>
<td>716</td>
</tr>
<tr>
<td>LONG_MAX</td>
<td>1450, 1618</td>
</tr>
<tr>
<td>LONG_MIN</td>
<td>1450, 1618</td>
</tr>
<tr>
<td>lrand48()</td>
<td>261, 718</td>
</tr>
<tr>
<td>lrint()</td>
<td>719</td>
</tr>
<tr>
<td>lrintf()</td>
<td>719</td>
</tr>
<tr>
<td>lrintl()</td>
<td>719</td>
</tr>
<tr>
<td>lround()</td>
<td>721</td>
</tr>
<tr>
<td>lroundf()</td>
<td>721</td>
</tr>
<tr>
<td>lroundl()</td>
<td>721</td>
</tr>
<tr>
<td>lsearch()</td>
<td>723</td>
</tr>
<tr>
<td>lseek()</td>
<td>725</td>
</tr>
<tr>
<td>lstat()</td>
<td>727</td>
</tr>
<tr>
<td>L_ctermid</td>
<td>239</td>
</tr>
<tr>
<td>l_sysid</td>
<td>331</td>
</tr>
<tr>
<td>makecontext()</td>
<td>729</td>
</tr>
<tr>
<td>malloc()</td>
<td>731</td>
</tr>
<tr>
<td>manipulate signal sets</td>
<td>1348</td>
</tr>
<tr>
<td>mappings</td>
<td>776</td>
</tr>
<tr>
<td>MAP</td>
<td>16, 18</td>
</tr>
<tr>
<td>MAP_FAILED</td>
<td>776</td>
</tr>
<tr>
<td>MAP_FIXED</td>
<td>777</td>
</tr>
<tr>
<td>MAP_PRIVATE</td>
<td>392, 772, 776, 780, 812</td>
</tr>
<tr>
<td>MAP_SHARED</td>
<td>395, 772-773</td>
</tr>
<tr>
<td>max-data-size attribute</td>
<td>76, 916-917</td>
</tr>
<tr>
<td>MAX_CANON</td>
<td>397</td>
</tr>
<tr>
<td>MAX_INPUT</td>
<td>397</td>
</tr>
<tr>
<td>may</td>
<td>2</td>
</tr>
<tr>
<td>mblen()</td>
<td>733</td>
</tr>
<tr>
<td>mbreln()</td>
<td>735</td>
</tr>
<tr>
<td>mbtowc()</td>
<td>737</td>
</tr>
<tr>
<td>mbsinit()</td>
<td>739</td>
</tr>
<tr>
<td>mbsrtowcs()</td>
<td>740</td>
</tr>
<tr>
<td>mbstowcs()</td>
<td>742</td>
</tr>
<tr>
<td>mbtowc()</td>
<td>744</td>
</tr>
<tr>
<td>MB_CUR_MAX</td>
<td>733, 735, 737, 744, 1591, 1633</td>
</tr>
<tr>
<td>MC1</td>
<td>5</td>
</tr>
<tr>
<td>MC2</td>
<td>5</td>
</tr>
<tr>
<td>MCL</td>
<td>16</td>
</tr>
<tr>
<td>MCL_CURRENT</td>
<td>769</td>
</tr>
<tr>
<td>MCL_FUTURE</td>
<td>769</td>
</tr>
<tr>
<td>memccpy()</td>
<td>746</td>
</tr>
<tr>
<td>memchr()</td>
<td>747</td>
</tr>
<tr>
<td>memcmp()</td>
<td>748</td>
</tr>
<tr>
<td>memcpy()</td>
<td>749</td>
</tr>
<tr>
<td>MEMLOCK_FUTURE</td>
<td>776</td>
</tr>
<tr>
<td>memmove()</td>
<td>750</td>
</tr>
<tr>
<td>memory management</td>
<td>43</td>
</tr>
<tr>
<td>memory protection option</td>
<td>775</td>
</tr>
<tr>
<td>memset()</td>
<td>751</td>
</tr>
<tr>
<td>message catalog descriptor</td>
<td>295, 300, 305</td>
</tr>
<tr>
<td>message parts</td>
<td>39</td>
</tr>
<tr>
<td>message priority</td>
<td>39</td>
</tr>
<tr>
<td>high-priority</td>
<td>39</td>
</tr>
<tr>
<td>normal</td>
<td>39</td>
</tr>
<tr>
<td>priority</td>
<td>39</td>
</tr>
<tr>
<td>message-discard mode</td>
<td>1168</td>
</tr>
<tr>
<td>message-nondiscard mode</td>
<td>1168</td>
</tr>
<tr>
<td>MET-1MEST</td>
<td>1542</td>
</tr>
<tr>
<td>MF</td>
<td>5</td>
</tr>
<tr>
<td>MINSIGSTKSZ</td>
<td>1345</td>
</tr>
<tr>
<td>mkdir()</td>
<td>752</td>
</tr>
<tr>
<td>mkfifo()</td>
<td>755</td>
</tr>
<tr>
<td>mkod()</td>
<td>758</td>
</tr>
<tr>
<td>mktemp()</td>
<td>761</td>
</tr>
<tr>
<td>mktemp()</td>
<td>763</td>
</tr>
<tr>
<td>mtimew()</td>
<td>765</td>
</tr>
<tr>
<td>mlock()</td>
<td>767</td>
</tr>
<tr>
<td>mlockall()</td>
<td>769</td>
</tr>
<tr>
<td>MLR</td>
<td>771</td>
</tr>
<tr>
<td>mmap()</td>
<td>771</td>
</tr>
</tbody>
</table>
Index

MM_ ................................................................. 18
MM_APPL......................................................... 383
MM_CONSEL....................................................... 383
MM_ERROR ..................................................... 384-385
mm_FIRM......................................................... 383
MM_HALT......................................................... 384
MM_HARD......................................................... 383
MM_INFO......................................................... 384
MM_NOCON...................................................... 384
MM_NOMSG....................................................... 384
MM_NOSEV...................................................... 384
MM_NOTOK...................................................... 384
MM_NRECOV.................................................... 383
MM_NULLMC.................................................... 383
MM_OK............................................................ 384
MM_OPSYS...................................................... 383
MM_PRINT...................................................... 383, 385
MM_RECOVER.................................................. 383
MM_SOFT......................................................... 383
MM_UTIL......................................................... 383
MM_WARNING.................................................. 384
mod().............................................................. 778
modf()............................................................ 778
modfl()............................................................ 778
MON.............................................................. 6
MORECTL....................................................... 523
MOREDATA...................................................... 523
MPR.............................................................. 6
mprotect()...................................................... 780
MQ.............................................................. 16
mq............................................................... 16
mq_close()..................................................... 782
mq_getattr()................................................... 783
mq_notify()..................................................... 785
mq_open()...................................................... 787
MQ_OPEN_MAX.................................................. 1466
MQ_Prio_MAX.................................................. 793-794, 1466
mq_receive()................................................... 790
mq_send()....................................................... 793
mq_setattr()................................................... 795
mq_timedreceive().......................................... 790, 797
mq_timedsend()............................................... 793, 798
mq_unlink().................................................... 799
mrand48()...................................................... 261, 801
MSG.............................................................. 18
msgctl()......................................................... 802
msgget()......................................................... 804
msggrv()........................................................ 806
msgsnd()......................................................... 809
MSG_VERB...................................................... 384-385
msg .............................................................. 16
MSG_RAW ....................................................... 18
MSG_ANY........................................................ 522
MSG_BAND...................................................... 522, 1150
MSG_EOR....................................................... 1387, 1389
MSG_HIPRI..................................................... 522, 1150
MSG_NOERROR................................................ 806-807
msg_perm....................................................... 40
msqid............................................................. 40
MST7MDT....................................................... 1542
msync().......................................................... 812
MS ............................................................... 16, 18
MS_ASYNC..................................................... 773, 812
MS_INVALIDATE.............................................. 812-813
MS_SYNC...................................................... 773, 812
multicast....................................................... 67
munlock()....................................................... 767, 815
munlockall().................................................. 769, 816
munmap()....................................................... 817
mutex attributes............................................. 1088
mutex initialization attributes....................... 1087
mutex performance........................................ 1088
MUXID_ALL................................................... 618-619
MUXID_R....................................................... 18
MX............................................................... 6
M ................................................................. 18
name information.......................................... 526
name space.................................................... 14
NAME_MAX ..................................................... 188, 298, 317, 336, 389, 397
................................................................. 428, 447, 456, 671, 680, 727
................................................................. 752, 755, 759, 788, 799, 827
................................................................. 837, 842, 964, 1174, 1178, 1185
................................................................. 1210, 1218, 1249, 1257, 1324
................................................................. 1327, 1398, 1463, 1535, 1555
................................................................. 1563, 1565
NaN.............................................................. 122, 405, 469
NAN.............................................................. 405, 469
nan().............................................................. 819
nanf()............................................................ 819
nanl()............................................................ 819
nanosleep().................................................... 820
NDEBUG......................................................... 21, 127
nearbyint()...................................................... 822
nearbyintf().................................................... 822
nearbyintl()..................................................... 822
network interfaces........................................ 59
NEW_TIME..................................................... 282-283
nextafter()..................................................... 824
nextafterf().................................................... 824
nextafterl().................................................... 824
nexttoward()................................................ 824
Index

nexttowardf() ........................................... 824
nexttowardl() ........................................... 824
nftw() ..................................................... 826
NGROUPS_MAX ....................................... 511, 1466
nice() ..................................................... 829
NLSPATH ............................................... 169
NL ........................................................ 18
NL_ARGMAX ....................................... 192, 343, 466, 475
NL_CAT_LOCALE ................................ 169
nl_langinfo() ......................................... 831
nohup utility ............................................ 301
non-local jumps ........................................ 1366
non-volatile storage ................................ 450
nrand48() ............................................. 261, 833
ntohl() ................................................... 584, 834
ntohs() ................................................... 584, 834
NULL ...................................................... 213, 239, 246, 254, 259, 776, 1176
NUM_EMPL ........................................... 582
NZERO ..................................................... 540, 829
n ........................................................... 22
OB ......................................................... 6
obsolescent .............................................. 178
OF ........................................................ 6
OH ........................................................ 6
OLD_TIME ............................................ 282-283
open a file .............................................. 839
open a named semaphore ....................... 129
open a shared memory object .................. 1324
open() .................................................... 835
opendir() .............................................. 842
openlog() .............................................. 208, 845
OPEN_MAX ......................................... 169, 263, 278, 330, 389
.................................................. 428, 461, 505, 508, 519, 548, 590
.................................................. 787, 827, 837, 842, 853, 880, 883
.................................................. 1467, 1524
optarg ................................................. 529, 846
opterr .................................................... 529, 846
optind .................................................... 529, 846
opt

ADV ..................................................... 3
ALO ..................................................... 3
BAR ..................................................... 3
BE ....................................................... 4
CD ....................................................... 4
CPT ....................................................... 4
CS ....................................................... 4
FD ....................................................... 4
FR ....................................................... 4
FSC ..................................................... 5
IP6 ..................................................... 5
MC1 ..................................................... 5
MC2 ..................................................... 5
MF ..................................................... 5
ML ..................................................... 5
MLR .................................................... 6
MON .................................................... 6
MPR .................................................... 6
MSG .................................................... 6
MX ..................................................... 6
PIO ..................................................... 7
PS ....................................................... 7
RS ....................................................... 7
RTS ..................................................... 7
SD ....................................................... 7
SEM ..................................................... 7
SHM ..................................................... 7
SIO ..................................................... 8
SPI ..................................................... 8
SPN ..................................................... 8
SS ....................................................... 8
TCT ..................................................... 8
TEF ..................................................... 8
THR ..................................................... 8
TMO ..................................................... 8
TMR ..................................................... 9
TPI ..................................................... 9
TPP ..................................................... 9
TPS ..................................................... 9
TRC ..................................................... 9
TRI ..................................................... 9
TRL ..................................................... 9
TS ...................................................... 10
TSA ..................................................... 10
TSF ..................................................... 10
TSH ..................................................... 10
TSP ..................................................... 10
TS ...................................................... 10
TSS ..................................................... 10
TYM ..................................................... 10
UP ..................................................... 10
XSR ..................................................... 11
optopt ................................................... 529, 532, 846
optstring .............................................. 532
orphaned process group ......................... 308
O ......................................................... 18
O_constants
  used in open() .................................... 835
  used in posix_openpt() ......................... 870
O_ACCMODE ....................................... 327
O_APPEND ....................................... 41, 115, 244, 341, 835, 1649
O_CREAT ......................................... 228, 787-788, 799, 835-837
O_TRUNC ......................................... 1243, 1248, 1233-1325
O_SYSV ............................................ 107, 835-836, 1168, 1650
O_EXCL ........................................... 788, 835-836, 1248, 1233-1234

1678

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_attr_getstackaddr()</td>
<td>995</td>
</tr>
<tr>
<td>pthread_attr_getstacksize()</td>
<td>997</td>
</tr>
<tr>
<td>pthread_attr_init()</td>
<td>978, 998</td>
</tr>
<tr>
<td>pthread_attr_setdetachstate()</td>
<td>981, 999</td>
</tr>
<tr>
<td>pthread_attr_setguardsize()</td>
<td>983, 1000</td>
</tr>
<tr>
<td>pthread_attr_setinheritsched()</td>
<td>985, 1001</td>
</tr>
<tr>
<td>pthread_attr_setschedparam()</td>
<td>987, 1002</td>
</tr>
<tr>
<td>pthread_attr_setschedpolicy()</td>
<td>989, 1003</td>
</tr>
<tr>
<td>pthread_attr_setscope()</td>
<td>991, 1004</td>
</tr>
<tr>
<td>pthread_attr_setstack()</td>
<td>993, 1005</td>
</tr>
<tr>
<td>pthread_attr_setstackaddr()</td>
<td>995, 1006</td>
</tr>
<tr>
<td>pthread_attr_setstacksize()</td>
<td>997, 1007</td>
</tr>
<tr>
<td>pthread_barrierattr_destroy()</td>
<td>1012</td>
</tr>
<tr>
<td>pthread_barrierattr_getshared()</td>
<td>1014</td>
</tr>
<tr>
<td>pthread_barrierattr_init()</td>
<td>1012, 1016</td>
</tr>
<tr>
<td>pthread_barrierattr_setscope()</td>
<td>1014, 1017</td>
</tr>
<tr>
<td>pthread_barrier_destroy()</td>
<td>1008</td>
</tr>
<tr>
<td>pthread_barrier_init()</td>
<td>1008</td>
</tr>
<tr>
<td>PTHREAD_BARRIER_SERIAL_THREAD</td>
<td>1010</td>
</tr>
<tr>
<td>pthread_barrier_wait()</td>
<td>1010</td>
</tr>
<tr>
<td>pthread_cancel()</td>
<td>1018</td>
</tr>
<tr>
<td>PTHREAD_CANCELED</td>
<td>57, 1053</td>
</tr>
<tr>
<td>PTHREAD_CANCELASYNCHRONOUS</td>
<td>54</td>
</tr>
<tr>
<td>PTHREAD_CANCEL_DEFERRED</td>
<td>54, 1032, 1128</td>
</tr>
<tr>
<td>PTHREAD_CANCEL_DISABLE</td>
<td>54, 1128</td>
</tr>
<tr>
<td>PTHREAD_CANCEL_ENABLE</td>
<td>54, 1128</td>
</tr>
<tr>
<td>pthread_cleanup_pop()</td>
<td>1020</td>
</tr>
<tr>
<td>pthread_cleanup_push()</td>
<td>1020</td>
</tr>
<tr>
<td>pthread_condattr_destroy()</td>
<td>1037</td>
</tr>
<tr>
<td>pthread_condattr_getclock()</td>
<td>1039</td>
</tr>
<tr>
<td>pthread_condattr_getshared()</td>
<td>1041</td>
</tr>
<tr>
<td>pthread_condattr_init()</td>
<td>1037, 1043</td>
</tr>
<tr>
<td>pthread_condattr_setclock()</td>
<td>1039, 1044</td>
</tr>
<tr>
<td>pthread_condattr_setshared()</td>
<td>1041, 1045</td>
</tr>
<tr>
<td>pthread_cond_broadcast()</td>
<td>1025</td>
</tr>
<tr>
<td>pthread_cond_destroy()</td>
<td>1028</td>
</tr>
<tr>
<td>pthread_cond_init()</td>
<td>1028</td>
</tr>
<tr>
<td>PTHREAD_COND_INITIALIZER</td>
<td>1028</td>
</tr>
<tr>
<td>pthread_cond_signal()</td>
<td>1025, 1031</td>
</tr>
<tr>
<td>pthread_cond_timedwait()</td>
<td>1032</td>
</tr>
<tr>
<td>pthread_cond_wait()</td>
<td>1032</td>
</tr>
<tr>
<td>pthread_create()</td>
<td>1046</td>
</tr>
<tr>
<td>PTHREAD_CREATEDETACHED</td>
<td>30, 981</td>
</tr>
<tr>
<td>PTHREAD_CREATETOGETHER</td>
<td>30, 981, 1063</td>
</tr>
<tr>
<td>PTHREAD_DESTRUCTOR_ITERATIONS</td>
<td>1060</td>
</tr>
<tr>
<td></td>
<td>1065, 1468</td>
</tr>
<tr>
<td>pthread_detach()</td>
<td>1049</td>
</tr>
<tr>
<td>pthread_equal()</td>
<td>1051</td>
</tr>
<tr>
<td>pthread_exit()</td>
<td>1052</td>
</tr>
<tr>
<td>PTHREAD_EXPLICIT_SCHED</td>
<td>985</td>
</tr>
<tr>
<td>Function</td>
<td>Page Numbers</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>pthread_getconcurrency()</td>
<td>1054</td>
</tr>
<tr>
<td>pthread_getcpulockid()</td>
<td>1056</td>
</tr>
<tr>
<td>pthread_getschedparam()</td>
<td>1057</td>
</tr>
<tr>
<td>pthread_getspecific()</td>
<td>1060</td>
</tr>
<tr>
<td>PTHREAD_INHERIT_SCHED</td>
<td>985</td>
</tr>
<tr>
<td>pthread_join()</td>
<td>1062</td>
</tr>
<tr>
<td>PTHREAD_KEYS_MAX</td>
<td>1065, 1468</td>
</tr>
<tr>
<td>pthread_key_create()</td>
<td>1065</td>
</tr>
<tr>
<td>pthread_key_delete()</td>
<td>1069</td>
</tr>
<tr>
<td>pthread_kill()</td>
<td>1071</td>
</tr>
<tr>
<td>pthread_mutexattr_destroy()</td>
<td>1087</td>
</tr>
<tr>
<td>pthread_mutexattr_getprioceiling()</td>
<td>1092</td>
</tr>
<tr>
<td>pthread_mutexattr_getprotocol()</td>
<td>1094</td>
</tr>
<tr>
<td>pthread_mutexattr_getshared()</td>
<td>1096</td>
</tr>
<tr>
<td>pthread_mutexattr_gettype()</td>
<td>1098</td>
</tr>
<tr>
<td>pthread_mutexattr_init()</td>
<td>1087, 1100</td>
</tr>
<tr>
<td>pthread_mutexattr_setprioceiling()</td>
<td>1092, 1101</td>
</tr>
<tr>
<td>pthread_mutexattr_setprotocol()</td>
<td>1094, 1102</td>
</tr>
<tr>
<td>pthread_mutexattr_setshared()</td>
<td>1096, 1103</td>
</tr>
<tr>
<td>pthread_mutexattr_settype()</td>
<td>1098, 1104</td>
</tr>
<tr>
<td>PTHREAD_MUTEX_DEFAULT</td>
<td>1080, 1098</td>
</tr>
<tr>
<td>pthread_mutex_destroy()</td>
<td>1072</td>
</tr>
<tr>
<td>PTHREAD_MUTEX_ERRORCHECK</td>
<td>1080, 1098</td>
</tr>
<tr>
<td>pthread_mutexgetprioceiling()</td>
<td>1077</td>
</tr>
<tr>
<td>pthread_mutex_init()</td>
<td>1072, 1079</td>
</tr>
<tr>
<td>PTHREAD_MUTEX_INITIALIZER</td>
<td>1072, 1079</td>
</tr>
<tr>
<td>pthread_mutexlock()</td>
<td>1080</td>
</tr>
<tr>
<td>PTHREAD_MUTEX_NORMAL</td>
<td>1080, 1098</td>
</tr>
<tr>
<td>PTHREAD_MUTEX_RECURSIVE</td>
<td>1098-1099</td>
</tr>
<tr>
<td>pthread_mutexsetprioceiling()</td>
<td>1077</td>
</tr>
<tr>
<td>pthread_mutex_timedlock()</td>
<td>1084</td>
</tr>
<tr>
<td>pthread_mutex_trylock()</td>
<td>1080, 1086</td>
</tr>
<tr>
<td>pthread_mutex_unlock()</td>
<td>1080, 1086</td>
</tr>
<tr>
<td>pthread_once()</td>
<td>1105</td>
</tr>
<tr>
<td>PTHREAD_ONCE_INIT</td>
<td>1105</td>
</tr>
<tr>
<td>PTHREAD_PRIO_INHERIT</td>
<td>1094</td>
</tr>
<tr>
<td>PTHREAD_PRIO_NONE</td>
<td>1094</td>
</tr>
<tr>
<td>PTHREAD_PRIO_PROTECT</td>
<td>1081, 1094</td>
</tr>
<tr>
<td>PTHREAD_PROCESS_PRIVATE</td>
<td>1014, 1041</td>
</tr>
<tr>
<td>PTHREAD_PROCESS_SHARED</td>
<td>1014, 1041</td>
</tr>
<tr>
<td>PTHREAD_PROCESS_SHARED</td>
<td>1088, 1096, 1123, 1137</td>
</tr>
<tr>
<td>pthread_rwlockattr_destory()</td>
<td>1121</td>
</tr>
<tr>
<td>pthread_rwlockattr_getprioceiling()</td>
<td>1123</td>
</tr>
<tr>
<td>pthread_rwlockattr_init()</td>
<td>1121, 1125</td>
</tr>
<tr>
<td>pthread_rwlockattr_setprioceiling()</td>
<td>1123, 1126</td>
</tr>
<tr>
<td>pthread_rwlock_destroy()</td>
<td>1107</td>
</tr>
<tr>
<td>pthread_rwlock_init()</td>
<td>1107</td>
</tr>
<tr>
<td>pthread_rwlock_rdlock()</td>
<td>1109</td>
</tr>
<tr>
<td>pthread_rwlock_timedrdlock()</td>
<td>1111</td>
</tr>
<tr>
<td>pthread_rwlock_timedwrlock()</td>
<td>1113</td>
</tr>
<tr>
<td>pthread_rwlock_tryrdlock()</td>
<td>1109, 1115</td>
</tr>
<tr>
<td>pthread_rwlock_trywrlock()</td>
<td>1116</td>
</tr>
<tr>
<td>pthread_rwlock_unlock()</td>
<td>1118</td>
</tr>
<tr>
<td>pthread_rwlock_wrlock()</td>
<td>1116, 1120</td>
</tr>
<tr>
<td>PTHREAD_SCOPE_PROCESS</td>
<td>52-53, 991</td>
</tr>
<tr>
<td>PTHREAD_SCOPE_SYSTEM</td>
<td>52-53, 991</td>
</tr>
<tr>
<td>pthread_self()</td>
<td>1127</td>
</tr>
<tr>
<td>pthread_setcancelstate()</td>
<td>1128</td>
</tr>
<tr>
<td>pthread_setcanceltype()</td>
<td>1128</td>
</tr>
<tr>
<td>pthread_setconcurrency()</td>
<td>1054, 1130</td>
</tr>
<tr>
<td>pthread_setschedparam()</td>
<td>1057, 1131</td>
</tr>
<tr>
<td>pthread_setschedpriority()</td>
<td>1132</td>
</tr>
<tr>
<td>pthread_setspecific()</td>
<td>1060, 1134</td>
</tr>
<tr>
<td>pthread_sigmask()</td>
<td>1135</td>
</tr>
<tr>
<td>pthread_spin_destroy()</td>
<td>1137</td>
</tr>
<tr>
<td>pthread_spin_init()</td>
<td>1137</td>
</tr>
<tr>
<td>pthread_spin_lock()</td>
<td>1139</td>
</tr>
<tr>
<td>pthread_spin_trylock()</td>
<td>1139</td>
</tr>
<tr>
<td>pthread_spin_unlock()</td>
<td>1141</td>
</tr>
<tr>
<td>PTHREAD_STACK_MIN</td>
<td>993, 995, 997, 1468</td>
</tr>
<tr>
<td>pthread_testcancel()</td>
<td>1128, 1142</td>
</tr>
<tr>
<td>PTHREAD_THREADS_MAX</td>
<td>1046, 1468</td>
</tr>
<tr>
<td>ptsname()</td>
<td>1143</td>
</tr>
<tr>
<td>putc()</td>
<td>1144</td>
</tr>
<tr>
<td>putchar()</td>
<td>1146</td>
</tr>
<tr>
<td>putchar_unlocked()</td>
<td>485, 1147</td>
</tr>
<tr>
<td>putc_unlocked()</td>
<td>485, 1145</td>
</tr>
<tr>
<td>putenv()</td>
<td>1148</td>
</tr>
<tr>
<td>putmsg()</td>
<td>1150</td>
</tr>
<tr>
<td>putppnmsg()</td>
<td>1150</td>
</tr>
<tr>
<td>puts()</td>
<td>1154</td>
</tr>
<tr>
<td>putuxline()</td>
<td>282, 1156</td>
</tr>
<tr>
<td>putwc()</td>
<td>1157</td>
</tr>
<tr>
<td>putwchar()</td>
<td>1158</td>
</tr>
<tr>
<td>pwrites()</td>
<td>1649, 1159</td>
</tr>
<tr>
<td>pw_</td>
<td>16</td>
</tr>
<tr>
<td>p_</td>
<td>16</td>
</tr>
<tr>
<td>P_</td>
<td>17</td>
</tr>
<tr>
<td>P_ALL</td>
<td>1588</td>
</tr>
<tr>
<td>P_PGID</td>
<td>1588</td>
</tr>
<tr>
<td>P_PID</td>
<td>1588</td>
</tr>
<tr>
<td>qsort()</td>
<td>1160</td>
</tr>
<tr>
<td>queue a signal to a process</td>
<td>1364</td>
</tr>
<tr>
<td>raise()</td>
<td>1364</td>
</tr>
<tr>
<td>rand()</td>
<td>1161</td>
</tr>
<tr>
<td>random()</td>
<td>605, 1166</td>
</tr>
<tr>
<td>RAND_MAX</td>
<td>1163</td>
</tr>
<tr>
<td>rand_r()</td>
<td>1163</td>
</tr>
<tr>
<td>read from a file</td>
<td>1170</td>
</tr>
<tr>
<td>read()</td>
<td>1167</td>
</tr>
<tr>
<td>readdir()</td>
<td>1174</td>
</tr>
</tbody>
</table>
Index

real user ID ..........................................................1195
realloc() ............................................................1183
realpath() ...........................................................1185
REALTIME ............................................................104120, 106-107, 109
..............................................................112-113, 115, 195, 201, 335
..............................................................683, 767, 769, 782-783, 785
..............................................................787, 790, 793, 795, 799, 820
..............................................................1228-1232, 1235, 1243-1246
..............................................................1248, 1251, 1255, 1257, 1259
..............................................................1323, 1327, 1363, 1370, 1376
..............................................................1514, 1517-1518
REALTIME THREADS ............................................985, 989, 991
.............................................................1001, 1003-1004, 1057, 1077
....................................................1083, 1092, 1094, 1101-1102
..............................................................1131-1132
recv() ..................................................................1187
recvfrom() ................................................................1189
recvmsg() ................................................................1192
regcomp() ................................................................1195
regerror() ...............................................................1195
regexec() ................................................................1195
regfree() ................................................................1195
register fork handlers .............................................976
REG .................................................................18
REG_ constants..........................................................1197
  error return values of regcomp ................................1197
  used in regcomp ...................................................1195
REG_BADBR ............................................................1197
REG_BADPAT ............................................................1197
REG_BADRPT ............................................................1197
REG_EBRACE ............................................................1197
REG_EBRACK ............................................................1197
REG_RECVALL ............................................................1197
REGECTYPE ............................................................1197
REG_EESCAPE ............................................................1197
REG_EPAREN ............................................................1197
REG ERANGE ............................................................1197
REG_ESPACE ............................................................1197
REG_ESUBREG ............................................................1197
REG_EXTENDED ..........................................................1183
REG_ICASE ...............................................................1195
REG_EXTENDED ..........................................................1195
REG_NOTBOL ............................................................1196
REG_NOTEOL ............................................................1196
remainder() .............................................................1202
remainderf() ............................................................1202
remainderl() .............................................................1202
remove a directory ....................................................1219
remove directory entries ............................................1557
remove() ................................................................1204
remque() ................................................................1206
remquo() ................................................................1207
remquof() ................................................................1207
rename a file .............................................................1211
rename() ................................................................1209
rewind() ................................................................1213
rewinddir() ................................................................1214
r_ ................................................................................16
RE_DUP_MAX .............................................................1468
rindex() ...................................................................1215
rint() .........................................................................1216
RLIMIT .................................................................18
  RLIMIT AS .............................................................552
  RLIMIT Core ............................................................551
  RLIMIT CPU ............................................................551
  RLIMIT_DATA ..........................................................551
  RLIMITFSIZE ..........................................................551
  RLIMIT_NOFILE ......................................................551, 553
  RLIMIT_STACK ..........................................................551
  rlim_ ........................................................................16
  RLIM_ ........................................................................18
  RLIM_INFINITY ..........................................................551-552
  RLIM_SAVED_CUR ....................................................552
  RLIM_SAVED_MAX ....................................................552
rmdir() .....................................................................1218
RMSGD .................................................................613
RMSGN .................................................................613
RMSGD .................................................................613
RND ...........................................................................46
round robin ..............................................................46
round() .....................................................................1221
roundf() ..................................................................1221
roundl() ..................................................................1221
routines ..................................................................59
RPRTDAT ...............................................................613
RPRTDIS ...............................................................613
RPRTNORM ...............................................................613
RS ...............................................................................7
RS_HIPI .................................................................522, 612, 1150
RTLD .................................................................18
  RTLD_DEFAULT ..........................................................259
  RTLD_GLOBAL .......................................................252, 256-257, 260
  RTLD_LAZY .............................................................256, 259
  RTLD_LOCAL .............................................................257
  RTLD_NEXT .............................................................259-260
  RTLD NOW .............................................................256-257
  RTS .................................................................7

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved. 1683
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>setlogmask()</td>
<td>208, 1297</td>
</tr>
<tr>
<td>setnetent()</td>
<td>274, 1298</td>
</tr>
<tr>
<td>setpgid()</td>
<td>1299</td>
</tr>
<tr>
<td>setpgrp()</td>
<td>1301</td>
</tr>
<tr>
<td>setpriority()</td>
<td>540, 1302</td>
</tr>
<tr>
<td>setprotoent()</td>
<td>276, 1303</td>
</tr>
<tr>
<td>setpwnent()</td>
<td>278, 1304</td>
</tr>
<tr>
<td>setregid()</td>
<td>1305</td>
</tr>
<tr>
<td>setreuid()</td>
<td>1307</td>
</tr>
<tr>
<td>setrlimit()</td>
<td>551, 1309</td>
</tr>
<tr>
<td>setservent()</td>
<td>280, 1310</td>
</tr>
<tr>
<td>setsid()</td>
<td>1311</td>
</tr>
<tr>
<td>setsockopt()</td>
<td></td>
</tr>
<tr>
<td>setsockopt()</td>
<td></td>
</tr>
<tr>
<td>setstate()</td>
<td>605, 1316</td>
</tr>
<tr>
<td>setutxent()</td>
<td>282, 1320</td>
</tr>
<tr>
<td>shal()</td>
<td>2</td>
</tr>
<tr>
<td>shal()</td>
<td>2</td>
</tr>
<tr>
<td>shell()</td>
<td>300, 308, 520, 536, 666, 1300, 1385</td>
</tr>
<tr>
<td>shell()</td>
<td>666</td>
</tr>
<tr>
<td>shell()</td>
<td>520</td>
</tr>
<tr>
<td>shell scripts</td>
<td></td>
</tr>
<tr>
<td>exec()</td>
<td>300</td>
</tr>
<tr>
<td>exec()</td>
<td>300</td>
</tr>
<tr>
<td>SHM()</td>
<td>7, 18</td>
</tr>
<tr>
<td>SHM()</td>
<td>7, 18</td>
</tr>
<tr>
<td>shmat()</td>
<td>1329</td>
</tr>
<tr>
<td>shmat()</td>
<td>1329</td>
</tr>
<tr>
<td>shmdt()</td>
<td>1331</td>
</tr>
<tr>
<td>shmdt()</td>
<td>1331</td>
</tr>
<tr>
<td>shmget()</td>
<td>1333</td>
</tr>
<tr>
<td>shmget()</td>
<td>1333</td>
</tr>
<tr>
<td>shmid()</td>
<td>1334</td>
</tr>
<tr>
<td>shmid()</td>
<td>1334</td>
</tr>
<tr>
<td>SHMLBA()</td>
<td>1329</td>
</tr>
<tr>
<td>shm()</td>
<td>16</td>
</tr>
<tr>
<td>SHM()</td>
<td>18</td>
</tr>
<tr>
<td>sh_open()</td>
<td>1323</td>
</tr>
<tr>
<td>sh_open()</td>
<td>1323</td>
</tr>
<tr>
<td>shm_perm()</td>
<td>40</td>
</tr>
<tr>
<td>shm_perm()</td>
<td>40</td>
</tr>
<tr>
<td>SHM_RDONLY()</td>
<td>1329</td>
</tr>
<tr>
<td>SHM_RDONLY()</td>
<td>1329</td>
</tr>
<tr>
<td>shm_unlink()</td>
<td>1327</td>
</tr>
<tr>
<td>shm_unlink()</td>
<td>1327</td>
</tr>
<tr>
<td>should()</td>
<td>2</td>
</tr>
<tr>
<td>should()</td>
<td>2</td>
</tr>
<tr>
<td>shutdown()</td>
<td>1336</td>
</tr>
<tr>
<td>SHUT()</td>
<td>18</td>
</tr>
<tr>
<td>SIGABRT()</td>
<td>92</td>
</tr>
<tr>
<td>sigaction()</td>
<td>1338</td>
</tr>
<tr>
<td>sigaction()</td>
<td>1338</td>
</tr>
<tr>
<td>sigaddset()</td>
<td>1344</td>
</tr>
<tr>
<td>SIGALRM()</td>
<td>118, 517, 1382, 1544, 1561</td>
</tr>
<tr>
<td>sigaltstack()</td>
<td>1345</td>
</tr>
<tr>
<td>SIGBUS()</td>
<td>44, 773, 776, 1135</td>
</tr>
<tr>
<td>SIGCANCEL()</td>
<td>1018</td>
</tr>
<tr>
<td>SIGCHLD()</td>
<td>209, 305-306, 554, 579, 1339, 1342, 1351, 1474, 1582, 1588</td>
</tr>
<tr>
<td>SIGCHLD()</td>
<td>1339, 1342, 1351, 1474, 1582, 1588</td>
</tr>
<tr>
<td>SIGCONT()</td>
<td>34, 306, 308, 665-666</td>
</tr>
<tr>
<td>sigdelset()</td>
<td>1347</td>
</tr>
<tr>
<td>sigemptyset()</td>
<td>1348</td>
</tr>
<tr>
<td>SIGEV()</td>
<td>16</td>
</tr>
<tr>
<td>sigev()</td>
<td>16</td>
</tr>
<tr>
<td>SIGEV_NONE()</td>
<td>29, 42</td>
</tr>
<tr>
<td>SIGEV_SIGNAL</td>
<td>29, 1514</td>
</tr>
<tr>
<td>SIGEV_THREAD</td>
<td>30</td>
</tr>
<tr>
<td>sigfillset()</td>
<td>1350</td>
</tr>
<tr>
<td>SIGFPE()</td>
<td>1135, 1357</td>
</tr>
<tr>
<td>sighold()</td>
<td>1351</td>
</tr>
<tr>
<td>SIGHUP()</td>
<td>203, 306, 308</td>
</tr>
<tr>
<td>sigignore()</td>
<td>1351</td>
</tr>
<tr>
<td>SIGILL()</td>
<td>1135, 1357</td>
</tr>
<tr>
<td>SIGINT()</td>
<td>394, 1474</td>
</tr>
<tr>
<td>siginterrupt()</td>
<td>1353</td>
</tr>
<tr>
<td>sigismember()</td>
<td>1355</td>
</tr>
<tr>
<td>SICKILL()</td>
<td>666, 1338, 1342, 1351</td>
</tr>
<tr>
<td>siglongjmp()</td>
<td>1356</td>
</tr>
<tr>
<td>signal generation and delivery</td>
<td>28</td>
</tr>
<tr>
<td>real time</td>
<td>29</td>
</tr>
<tr>
<td>signal handler</td>
<td>1357</td>
</tr>
<tr>
<td>signal()</td>
<td>1357</td>
</tr>
<tr>
<td>signaling a condition</td>
<td>1026</td>
</tr>
<tr>
<td>signals</td>
<td>28</td>
</tr>
<tr>
<td>signbit()</td>
<td>1359</td>
</tr>
<tr>
<td>sigpause()</td>
<td>1351, 1360</td>
</tr>
<tr>
<td>sigpending()</td>
<td>1361</td>
</tr>
<tr>
<td>SIGPIPE()</td>
<td>325, 358, 414, 418, 441, 444, 1151, 1652</td>
</tr>
<tr>
<td>SIGPOLL()</td>
<td>203, 611-612</td>
</tr>
<tr>
<td>sigprocmask()</td>
<td>1135, 1362</td>
</tr>
<tr>
<td>SIGPROF()</td>
<td>517</td>
</tr>
<tr>
<td>sigqueue()</td>
<td>1363</td>
</tr>
<tr>
<td>SIGQUEUE_MAX</td>
<td>1363, 1468</td>
</tr>
<tr>
<td>SIGQUIT()</td>
<td>1474</td>
</tr>
<tr>
<td>sigrelse()</td>
<td>1351, 1365</td>
</tr>
<tr>
<td>SIGRTMAX()</td>
<td>29-30, 1363, 1370, 1374</td>
</tr>
<tr>
<td>SIGRTMIN()</td>
<td>29-30, 1363, 1370, 1374</td>
</tr>
<tr>
<td>SIGSEGV()</td>
<td>44, 552, 817, 983, 1135, 1357</td>
</tr>
<tr>
<td>sigset()</td>
<td>1351, 1365</td>
</tr>
<tr>
<td>sigsetjmp()</td>
<td>1366</td>
</tr>
<tr>
<td>SIGSTKSZ()</td>
<td>1345</td>
</tr>
<tr>
<td>SIGSTOP()</td>
<td>29, 1338, 1342, 1351</td>
</tr>
<tr>
<td>sigsuspend()</td>
<td>1368</td>
</tr>
<tr>
<td>sigtimedwait()</td>
<td>1370</td>
</tr>
<tr>
<td>SIGTSTP()</td>
<td>29</td>
</tr>
<tr>
<td>SIGTTIN()</td>
<td>29, 362, 368, 1169</td>
</tr>
<tr>
<td>SIGTTOU()</td>
<td>29, 325, 358, 414, 418</td>
</tr>
<tr>
<td></td>
<td>441, 443, 1483, 1485, 1487, 1494</td>
</tr>
<tr>
<td></td>
<td>1497, 1652</td>
</tr>
</tbody>
</table>

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
SIGURG............................................................612
SIGVTAALRM.....................................................517
sigwait() ..........................................................1374
sigwaitinfo() ....................................................1370, 1376
SIGXCPU..........................................................551
SIGXFSZ..........................................................551, 1535
SIG.................................................................16, 18
SIG_BLOCK........................................................1135
SIG_DFL.........................................................31, 295, 552, 1338, 1340, 1357
SIG_ERR..........................................................147, 1357
SIG_HOLD..........................................................1351
SIG_IGN.........................................................31, 295, 301, 305-306, 554
.................................................................1338, 1357, 1382
SIG_SETMASK....................................................1135
SIG_UNBLOCK....................................................1135
sin() .................................................................1377
sin6()...............................................................16
sinf() ...............................................................1377
sinh()..............................................................1379
sinh2().............................................................1379
sinh1().............................................................1379
sinl()..............................................................1377, 1381
sin.................................................................16
SION...............................................................8
SIOCATMARK.....................................................1385
sival...............................................................16
si.................................................................16-17
SL.................................................................16, 18
SIASYNCIO........................................................32
SI_MSGQ..........................................................32
SIQUEUE..........................................................32
SI_TIMER........................................................32
SI_USER..........................................................32
sleep()............................................................1382
sl.................................................................16
SND...............................................................18
SNDZERO..........................................................615
snprintf().......................................................402, 1384
SO.................................................................18
socketmark()....................................................1385
socket I/O mode.................................................60
socket out-of-band data.....................................61
socket owner....................................................60
socket queue limits..........................................60
socket receive queue.......................................61
socket types....................................................59
socket()..........................................................1387
socketpair()....................................................1389
sockets...........................................................58
address families..............................................58
addressing......................................................59
asynchronous errors........................................62
connection indication queue...........................62
Internet Protocols...........................................66
IPv4...............................................................67
IPv6...............................................................67
local UNIX connections..................................66
options...........................................................63
pending error...................................................60
protocols.........................................................59
signals...........................................................62
SOCK.............................................................18
SOCK_DGRAM....................................................66, 1387, 1389
SOCK_RAW.......................................................66
SOCK_Sequence..............................................66, 1387, 1389
SOCK_STREAM...................................................66, 1387, 1389
SPI.................................................................8
SPN...............................................................8
sporadic server policy
  execution capacity........................................47
  replenishment period......................................47
sprintf()..........................................................402, 1391
spurious wakeup..............................................1026
sqrt()..............................................................1392
sqrtf().............................................................1392
sqrtl().............................................................1392
srand()...........................................................1163, 1394
srand48().......................................................261, 1395
srandom()......................................................605, 1396
SS.................................................................8
sscanf()..........................................................433, 1397
SSIZE_MAX...........................790, 806, 1167, 1178, 1419, 1649
ss.................................................................16-17
SS.................................................................18
SS_DISABLE......................................................1345-1346
SS_ONSTACK....................................................1345
stack size........................................................978
stat().............................................................1398
statvfs()..........................................................447, 1402
stderr..........................................................1403
STDERR_FILENO.................................................1403
stdin............................................................1403
STDIN_FILENO...................................................859, 1403
stdio locking functions....................................373
stdio with explicit client locking.......................485
stdout..........................................................1403
STDOUT_FILENO.................................859, 1403
STR...............................................................18
stracecmp()....................................................1405
strcat()..........................................................1406
strchr()..........................................................1407
strcmp().........................................................1408
Index
Index

strcoll() .................................................................1410
strcpy() ....................................................................1412
strncpy() .................................................................1414
strdup() .................................................................1415
STREAM .................................................................614, 616, 1150, 1168, 1651
stream
  byte-oriented .........................................................37
  wide-oriented .........................................................37
STREAM head/tail ......................................................38
stream-full-policy attribute .................................72-73, 76, 914
stream-min-size attribute ........................................76, 917
streams .................................................................34
  interaction with file descriptors ......................35
  stream orientation ..................................................37
STREAMS.................................................................22, 203, 317, 336, 522
                         .........................................................610, 625, 837-838, 855, 971
access .................................................................39
  multiplexed ..........................................................618
  overview ...............................................................38
STREAM_MAX ...........................................................340, 389, 859, 1468
strerror() .................................................................1416
strerror_r() .............................................................1416
strftime() .................................................................1418
strptime() .................................................................1422
strlen() .................................................................1427
strncasecmp() ........................................................1405, 1429
strncat() .................................................................1430
strncpy() .................................................................1431
strncpy() .................................................................1432
strncpy() .................................................................1433
strncpy() .................................................................1434
strchr() .................................................................1438
strchr() .................................................................1438
strncat() .................................................................1439
strncat() .................................................................1440
strtod() .................................................................1441
strtof() .................................................................1441
strtoimax() ............................................................1445
strtok() .................................................................1446
strtok_r() ...............................................................1446
strtol() .................................................................1449
strtold() .................................................................1441, 1451
strtoIl() .................................................................1449, 1452
strtozl() .................................................................1453
strtozl() .................................................................1453
strtozumax() ...........................................................1445, 1456
strxfrm() ...............................................................1457
str_ .................................................................1457
ST .................................................................1457
ST NOSUID .............................................................295, 447
ST_RDONLY ............................................................447
sun_ .................................................................1410
superuser .............................................................97, 189, 682, 1557
supplementary groups ...........................................189, 510
SVID .................................................................1366
SVR4 .................................................................775, 820
sv .................................................................1410
SV .................................................................1410
swab() .................................................................1459
swapcontext() ........................................................729, 1460
swprintf() ............................................................466, 1461
swerase() ...............................................................475, 1462
SWITCH ...............................................................1463
symbols
  POSIX.1 ...............................................................13
  symfony() ............................................................1463
SYMLINK_MAX ...........................................................397, 1463
SYMLINK_MAX ...........................................................397, 1463
SYMLOOP_MAX ......................................................146, 218, 318, 336, 389
                         .........................................................429, 448, 456, 462, 671, 759
                         .........................................................827, 1185, 1275, 1278, 1468
                         .........................................................1536, 1565
sync() .................................................................1465
synchronous accept a signal ..................................1371
sysconf() .............................................................1466
sylog() .................................................................208, 1473
System crash ..........................................................450
System III .............................................................189, 1550
  system interfaces ...................................................83
  system name ........................................................1550
  system trace event type definitions ......................76
System V ..............................................................118, 189, 303, 308, 330-331
                         .........................................................399, 536, 666, 753, 1219, 1311
                         .........................................................1342, 1366, 1489, 1550
system() ............................................................1474
S .................................................................1474
S BANDURG ...........................................................612
S ERROR ...............................................................611
S HANGUP .............................................................612
S HIPRI ...............................................................611
S IBBLK ...............................................................758
S IFCR ...............................................................758
S IFDIR ...............................................................758
S IFIFO ...............................................................758
S IFREG ...............................................................758
S INPUT ...............................................................611
S IRGRP ..............................................................321, 445, 758
S IROTH ..............................................................321, 445, 758
S IRUSR ..............................................................321, 445, 758
S IRWXG .............................................................758
S IRWXO .............................................................758
S IRWXU .............................................................758

System Interfaces, Issue 6 — Copyright © 2001, IEEE and The Open Group. All rights reserved.
<table>
<thead>
<tr>
<th>trace event, POSIX_TRACE_FILTER</th>
<th>77, 947</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace event, POSIX_TRACE_OVERFLOW</td>
<td>77</td>
</tr>
<tr>
<td>trace event, POSIX_TRACE_RESUME</td>
<td>77</td>
</tr>
<tr>
<td>trace event, POSIX_TRACE_START</td>
<td>77, 956</td>
</tr>
<tr>
<td>trace event, POSIX_TRACE_STOP</td>
<td>77, 956</td>
</tr>
<tr>
<td>trace functions</td>
<td>79</td>
</tr>
<tr>
<td>trace-name attribute</td>
<td>76, 911</td>
</tr>
<tr>
<td>TRACE_EVENT_NAME_MAX</td>
<td>936, 938</td>
</tr>
<tr>
<td>TRACE_SYS_MAX</td>
<td>933</td>
</tr>
<tr>
<td>TRACE_USER_EVENT_MAX</td>
<td>936, 938</td>
</tr>
<tr>
<td>TRACING</td>
<td>909, 911, 913, 916, 919-928</td>
</tr>
<tr>
<td></td>
<td>930, 932, 936, 938, 940-941</td>
</tr>
<tr>
<td></td>
<td>943-945, 947, 949-950, 953-956</td>
</tr>
<tr>
<td>TRAP</td>
<td>948-960</td>
</tr>
<tr>
<td>TRC</td>
<td>18</td>
</tr>
<tr>
<td>TRI</td>
<td>9</td>
</tr>
<tr>
<td>TRL</td>
<td>9</td>
</tr>
<tr>
<td>truncate()</td>
<td>1534</td>
</tr>
<tr>
<td>trunc()</td>
<td>1535</td>
</tr>
<tr>
<td>truncation-status attribute</td>
<td>936</td>
</tr>
<tr>
<td>trunct()</td>
<td>1534, 1537</td>
</tr>
<tr>
<td>TSC</td>
<td>10</td>
</tr>
<tr>
<td>tsearch()</td>
<td>1501, 1538</td>
</tr>
<tr>
<td>TSF</td>
<td>10</td>
</tr>
<tr>
<td>TSH</td>
<td>10</td>
</tr>
<tr>
<td>TSP</td>
<td>10</td>
</tr>
<tr>
<td>TSS</td>
<td>10</td>
</tr>
<tr>
<td>ttyname()</td>
<td>1539</td>
</tr>
<tr>
<td>ttyname_r()</td>
<td>1539</td>
</tr>
<tr>
<td>TTY_NAME_MAX</td>
<td>1468, 1539</td>
</tr>
<tr>
<td>tv_</td>
<td>16-17</td>
</tr>
<tr>
<td>twalk()</td>
<td>1501, 1541</td>
</tr>
<tr>
<td>TYM</td>
<td>10</td>
</tr>
<tr>
<td>tznname()</td>
<td>1542</td>
</tr>
<tr>
<td>TZNAME_MAX</td>
<td>1468</td>
</tr>
<tr>
<td>tzset()</td>
<td>1542, 1542</td>
</tr>
<tr>
<td>t_uscalar_t</td>
<td>613</td>
</tr>
<tr>
<td>utalarm()</td>
<td>1544</td>
</tr>
<tr>
<td>uc_</td>
<td>16-17</td>
</tr>
<tr>
<td>UINT</td>
<td>18</td>
</tr>
<tr>
<td>UINT_MAX</td>
<td>118, 1383</td>
</tr>
<tr>
<td>UIO_MAXIVO</td>
<td>17</td>
</tr>
<tr>
<td>ulimit()</td>
<td>1546</td>
</tr>
<tr>
<td>ULLONG_MAX</td>
<td>1454</td>
</tr>
<tr>
<td>ULONG_MAX</td>
<td>1454, 1625</td>
</tr>
<tr>
<td>UL_</td>
<td>17</td>
</tr>
<tr>
<td>UL_GETFSIZE</td>
<td>1546</td>
</tr>
<tr>
<td>UL_SETFSIZE</td>
<td>1546</td>
</tr>
<tr>
<td>umask()</td>
<td>1548</td>
</tr>
<tr>
<td>uname()</td>
<td>1550</td>
</tr>
<tr>
<td>undefined()</td>
<td>2</td>
</tr>
<tr>
<td>underlying function</td>
<td>36</td>
</tr>
<tr>
<td>ungetc()</td>
<td>1552</td>
</tr>
<tr>
<td>ungetwc()</td>
<td>1553</td>
</tr>
<tr>
<td>unicast()</td>
<td>67</td>
</tr>
<tr>
<td>unlink()</td>
<td>1555</td>
</tr>
<tr>
<td>unlockpt()</td>
<td>1559</td>
</tr>
<tr>
<td>unsetenv()</td>
<td>1560</td>
</tr>
<tr>
<td>unspecified()</td>
<td>2</td>
</tr>
<tr>
<td>UP</td>
<td>10</td>
</tr>
<tr>
<td>US-ASCII</td>
<td>624</td>
</tr>
<tr>
<td>user ID</td>
<td></td>
</tr>
</tbody>
</table>
|                                | real and effective | 1307
|                                | setting real and effective | 1307
|                                | user trace event type definitions | 79
|                                | USER_PROCESS         | 282-283
|                                | usleep()             | 1561
|                                | UTC                  | 1542
|                                | utimes()             | 1563
|                                | utin_                | 17
|                                | ut_                  | 17
|                                | va_arg()             | 1567
|                                | va_copy()            | 1567
|                                | va_end()             | 1567
|                                | va_start()           | 1567
|                                | VDISCARD             | 18
|                                | VDSUSP               | 18
|                                | Version 7            | 118, 189, 666, 1550
|                                | vfork()              | 1568
|                                | vfprintf()           | 1570
|                                | vfreopen()           | 1571
|                                | vfwprintf()          | 1572
|                                | vfwscanf()           | 1573
|                                | VLIST                | 1501, 1541
|                                | VMLNEXT              | 18
|                                | VPRINT               | 18
|                                | VREPRINT             | 18
|                                | vscanf()             | 1571, 1575
|                                | vsscanf()            | 1571, 1577
|                                | VSTATUS              | 18
|                                | vsprintf()           | 1572, 1578
|                                | vsscanf()            | 1573, 1579
|                                | VWERASE              | 18
|                                | vfwprintf()          | 1572, 1580
|                                | vsfwscanf()          | 1573, 1581
|                                | wait for process termination | 1585
Index

wait for thread termination..............................................1063
wait() ........................................................................1582
waitid() ......................................................................1588
waiting on a condition..................................................1033
waitpid() ......................................................................1582, 1590
WARNING ......................................................................384
warning
  OB ...............................................................................6
  OF ...............................................................................6
WCONTINUED ..................................................................1582, 1588
wrtomb() .......................................................................1591
wcscat() .......................................................................1593
wcschr() .......................................................................1594
wcsncmp() ......................................................................1595
wcscoll() ......................................................................1596
wcsncpy() ......................................................................1597
wcsncmpy() ....................................................................1604
wcspsbrk() ....................................................................1605
wcschr() ......................................................................1606
wcsrtombs() ....................................................................1607
wcspn() .......................................................................1609
wcsstr() .......................................................................1610
wcssto(const) .............................................................1611
wcstol() ......................................................................1611
wcstolmax() ..................................................................1614
wcstok() ......................................................................1615
wcstok() ......................................................................1616
wcstold() ......................................................................1611, 1620
wcstoll() ......................................................................1613, 1621
wcstombs() ....................................................................1622
wcstoul() .......................................................................1624
wcstoumax() ..................................................................1614, 1627
wcswcs() .......................................................................1628
wcswidth() ....................................................................1629
wcswxfrm() ..................................................................1630
wcstob() ........................................................................1632
wcstomb() ......................................................................1633
wctrans() ......................................................................1635
wctype() .........................................................................1636
wcwidth() .......................................................................1638
WEOF ............................................................................82, 647-648, 650-651, 653-660
                                  .............................................................1532-1533, 1553
WEXITED ........................................................................1588
WEXITSTATUS ................................................................1583
we_ ..................................................................................17
wide-oriented stream .....................................................37

WIFCONTINUED ...........................................................1583
WIFEXITED ...............................................................1583
WIFSIGNALED ..........................................................1583
WIFSTOPPED ..................................................................1583, 1586
wmemchr() .....................................................................1639
wmemcmp() .....................................................................1640
wmemcpy() .....................................................................1641
wmemmove() ....................................................................1642
wmemset() ........................................................................1643
WNOHANG .....................................................................1342, 1582, 1588
WNOWAIT .........................................................................1588
wordexp() .......................................................................1644
wfree() .............................................................................1644
wprintf() ......................................................................466, 1648
WRDE .............................................................................18
WRDE_APPEND ................................................................1645
WRDE_BADCHAR ................................................................1646
WRDE_BADVAL ................................................................1646
WRDE_CMDSUB ................................................................1646
WRDE_DOOFFS ................................................................1645
WRDE_NOCMD ................................................................1645
WRDE_NOSPACE ................................................................1646
WRDE_REUSE ..................................................................1645
WRDE_SHOWERR ................................................................1645
WRDE_SYNTAX ................................................................1646
WRDE_UNDEF ..................................................................1645
write to a file .................................................................1653
write() ...........................................................................1649
wscanf() ...........................................................................475, 1659
X_OK ...............................................................................97
XSI .............................................................. 39
XSI interprocess communication ...................................... 39
XSR .................................................................................11
Y_OK .................................................................................97
y0() .............................................................................1660
y1() .............................................................................1660
yn() .............................................................................1660
zombie process ..................................................................305